Computational Approaches to the Visual Validation of 3D Virtual Environments

Alfredo Nantes
Laurea (BSc + MSc) in Computer Engineering

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Faculty of Science and Technology
Queensland University of Technology

October 2011

© Copyright by Alfredo Nantes 2011
All rights reserved

Declaration

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signature:

Date:

Keywords

Virtual Environment Testing, Virtual Environment Validation, Virtual Environment Consistency, Rendering System Validation, Computer Game Testing, Visual Bug Detection, Visual Inconsistency Detection, Synthetic Image Processing.

Abstract

Virtual environments can provide, through digital games and online social interfaces, extremely exciting forms of interactive entertainment. Because of their capability in displaying and manipulating information in natural and intuitive ways, such environments have found extensive applications in decision support, education and training in the health and science domains amongst others.

Currently, the burden of validating both the interactive functionality and visual consistency of a virtual environment content is entirely carried out by developers and play-testers. While considerable research has been conducted in assisting the design of virtual world content and mechanics, to date, only limited contributions have been made regarding the automatic testing of the underpinning graphics software and hardware.

The aim of this thesis is to determine whether the correctness of the images generated by a virtual environment can be quantitatively defined, and automatically measured, in order to facilitate the validation of the content. In an attempt to provide an environment-independent definition of visual consistency, a number of classification approaches were developed.

First, a novel model-based object description was proposed in order to enable reasoning about the color and geometry change of virtual entities during a play-session. From such an analysis, two view-based connectionist approaches were developed to map from geometry and color spaces to a single, environment-independent, geometric transformation space; we used such a mapping to predict the correct visualization of the scene. Finally, an appearance-based aliasing detector was developed to show how incorrectness too, can be quantified for debugging purposes.

Since computer games heavily rely on the use of highly complex and interactive virtual worlds, they provide an excellent test bed against which to develop, calibrate and validate our techniques. Experiments were conducted on a game engine and other virtual worlds prototypes to determine the applicability and effectiveness of our algorithms. The results show that quantifying visual correctness in virtual scenes is a feasible enterprise, and that effective automatic bug detection can be performed through the techniques we have developed. We expect these techniques to find application in large 3D games and virtual world studios that require a scalable solution to testing their virtual world software and digital content.

Publications

1. Towsey, M., Planitz, B., Nantes, A., Wimmer, J., & Roe, P. (2011). A Toolbox for Animal Call Recognition. In *Bioacoustics*. Impact factor: 1.44. In Print.
2. Nantes, A. and Brown, R. A. & Maire, F. D. (2010). *Measuring Visual Consistency in 3D Rendering Systems*. Paper presented at the Thirty-Third Australian Computer Science Conference (ACSC-2010), Queensland University of Technology, Brisbane.
3. Pham, B. L., Zhang, J., & Nantes, A. (2009). Semantic and context-based retrieval of digital cultural objects [Published in Chinese]. In X. Zhang & M. A. Keane (Eds.), *International Perspectives on the Creative Economy* (Vol. 1, pp. 205-229): Sunchime Publishing.
4. Nantes, A., Brown, R., & Maire, F. D. (2008). *A Framework for the Semi-Automatic Testing of Video Games*. Paper presented at the Artificial Intelligence and Interactive Digital Entertainment (AIIDE-08), Stanford University, Palo Alto, California.

To Alex, Vera, Lory and daddy

Acknowledgements

I am greatly indebted to my supervisor Ross Brown for his patience, guidance and support over the course of this PhD journey. Ross continuously encouraged and trusted me in all my experimental endeavours, and helped me with everything I needed in order to complete and consolidate my work. The interest he showed in the material, his patience with my many delays and his willingness to listen to all my phenomenological digressions were beyond the call of duty. It is from Ross I learned that great work requires passion, creativity and dedication, and a stationary target.

I am also grateful to Frederic Maire, for his invaluable help and advice for any technical difficulty I experienced during this project. His extensive mathematical background and brilliant insights have always been a constant source of motivation and inspiration for me. Frederic is one of those minds who see obvious solutions to supposedly impossible problems. Luckily for me, he was my associate supervisor.

Thanks to the entire BPM group; to Michael Rosemann in particular, for the prompt and unquestioned support to this non-IS member of the IS discipline; to Jan Recker and Marcello La Rosa, for being inspirational to many of my important decisions; to Stephan Clemens and Thomas Kohlborn, for their entertaining and optimistic attitude that gave my stressful days a positive twist.

I am also grateful to Michael Towsey and Paul Roe, for offering me to work with them on their ambitious Sensor Networks project. Their incessant enthusiasm and dedication to rigorous research influenced my own work. I will go so far as to claim that this dissertation would have taken a different shape, had I not made Michael's acquaintance. I should say the same goes for Christian Flender, the most knowledgeable person in matters of philosophy of science I have ever encountered. Christian deeply influenced my own thinking about mind, the universe and everything.

Thanks to Chiara, Ian, Jesse, Julie, Sam and Andrea for showing me that life is immeasurably easier if you are well equipped with amazing friends.

Special thanks go to my family, my ever present supporters, for their permanent understanding of my mood swings and for being avid listeners of all my failures and successes. Anything good that I am, I owe to my family. They have always been the epitome of perseverance, sacrifice and love.

Glossary

χ^2	Chi-Square Distribution
AI	Artificial Intelligence
AIFD	Affine Invariant Fourier Descriptor
ANN	Artificial Neural Network
ANOVA	Analysis of Variance
API	Application Programming Interface
AUC	Area Under Curve
BMU	Best Matching Unit
BN	Bayesian Network
COM	Component Object Model
CPU	Central Processing Unit
CSS	Curvature Scale Space
DCT	Discrete Cosine Transform
DLL	Dynamic Link Library
DoG	Difference of Gaussian
FN	False Negative
FP	False Positive
FPR	False Positive Ratio
GHTD	Generalized Hough Transform Descriptor
GLOH	Gradient Location and Orientation Histogram
GMM	Gaussian Mixture Model
GPU	Graphics Processing Unit
HMM	Hidden Markov Model
HN	Hopfield Network
HOT	High Order Tangents
HSV	Cylindrical-coordinate representations of points in an RGB color model. HSV stands for Hue, Saturation, and Value
ICA	Independent Component analysis
IQR	Inter Quartile Range
kNN	k-Nearest Neighbor
LOR	Level of Realism
MGD	Multivariate Gaussian Distribution
MLP	Multi-Layer Perceptrons
MST	Minimum Spanning Tree
NDC	Normalized Device Coordinates
NMF	Non-negative Matrix Factorization
NPC	Non-Player Character
OBJ	Object or Geometry
PCA	Principal Component Analysis
QA	Quality Assurance
RAM	Random-Access Memory
RBF	Radial Basis Function
RGB	Color space defined by the three chromaticities of the red (R), green (G), and blue (B) additive primaries
RNN	Replicator Neural Network
ROC	Receiver Operating Characteristic
SDK	Software Development Kit
SIFT	Scale-Invariant Feature Transform
SOM	Self-Organizing Map
SPSD	Symmetric Positive Semidefinite
SSE	Sum Squared Error
SURF	Speeded Up Robust Features
SVM	Support Vector Machine
TN	True Negative
TP	True Positive
TPR	True Positive Ratio

GLOSSARY

Contents

List of Figures

List of Tables

1

Introduction

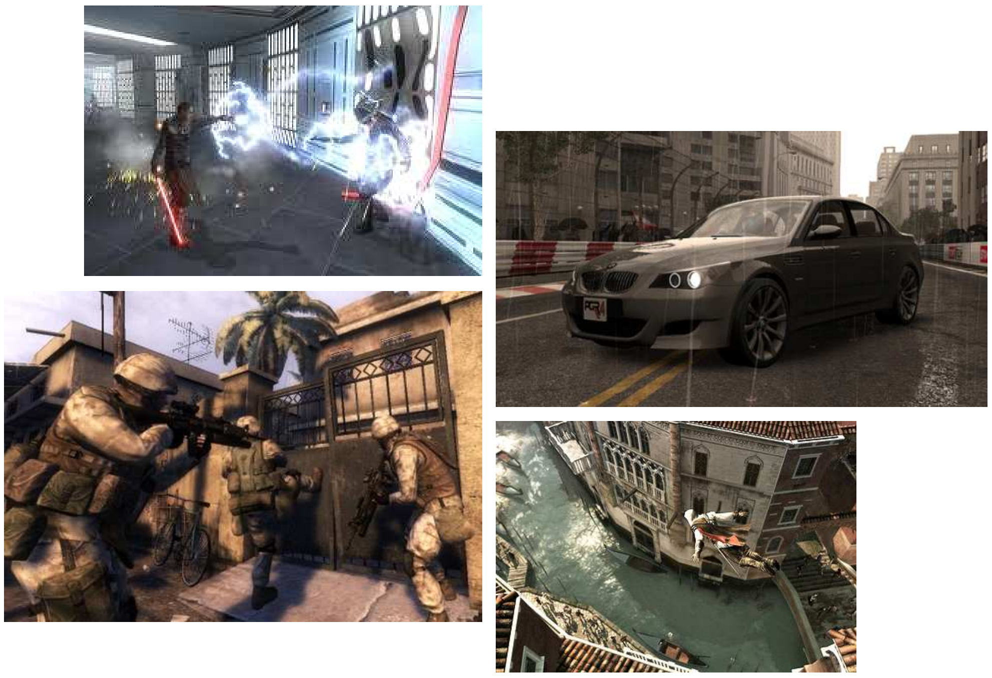


Figure 1.1: Modern 3D Virtual Environments

1.1 Motivation

Virtual environment technology today covers a large component of the entire entertainment industry, especially in the area of video games [5]. Present game worlds differ from early generations where the graphics consisted of very few polygons and the user-game interaction was restricted to a few commands entered through the keyboard. Today, virtual environments are composed of thousands of objects, defined with thousands of polygons [81] which gives the environment a remarkably realistic appearance. In such virtual worlds, a great deal of entities interact with each other in real time, according to complex physical models, or by following sophisticated logics. This advanced entertainment technology has emerged from years of ongoing research in, amongst other disciplines, computer graphics, artificial intelligence and computational physics [34].

In addition to entertainment, 3D virtual environments are used for therapeutic, educational and work-related purposes. Indeed, virtual reality has been successfully used to develop new diagnostic tools for psychology and neuro-psychology [64]; to facilitate surgical procedures, medical therapies and patient education [122]; and for distance education and collaboration purposes [7].

A game environment[1] is synthesized by a 3D application, which in turn, is a complex collection of software and hardware components that work in orchestration to produce the experience of a game (Figure 1.2). The most common element that a game application provides is graphics rendering facilities. The graphics engine includes scene graph management routines, geometry partitioning and search algorithms, and a graphics pipeline. Realistic interactive game environments also include, amongst others, system engines (containing memory and CPU management libraries); sound engines (enabling real-time audio mixing and filtering); AI modules (encoding the behaviour of the virtual entities); and physics engines (implementing numerical solution to physics models).

The more complex the software, the more crucial the effort of the companies in testing their product in order to ensure high enough quality in terms of functionality, stability and robustness in general. Furthermore, in the case of computer games, the 3D application is not only expected to work properly, but it has to be, amongst other things, fun, challenging, realistic and well animated [51]. Assessing individual software modules for robustness and efficiency certainly contributes to meeting design requirements about specific game assets such as interface elements, menu screens, use cases, sound and visual effects. However, it cannot guarantee that the integration of such modules with other (externally manufactured) software and hardware will not introduce artifacts.

Testing is a crucially important step in the development of interactive virtual environments. Yet, there is no known best testing practice that can be followed to draw

[1]Computer games represent a specific application of virtual environment technology. However, so far as this work is concerned, both virtual environments and games can be regarded as mechanisms for generating visually realistic scenes. Thus, we shall use these two terms interchangeably in this document.

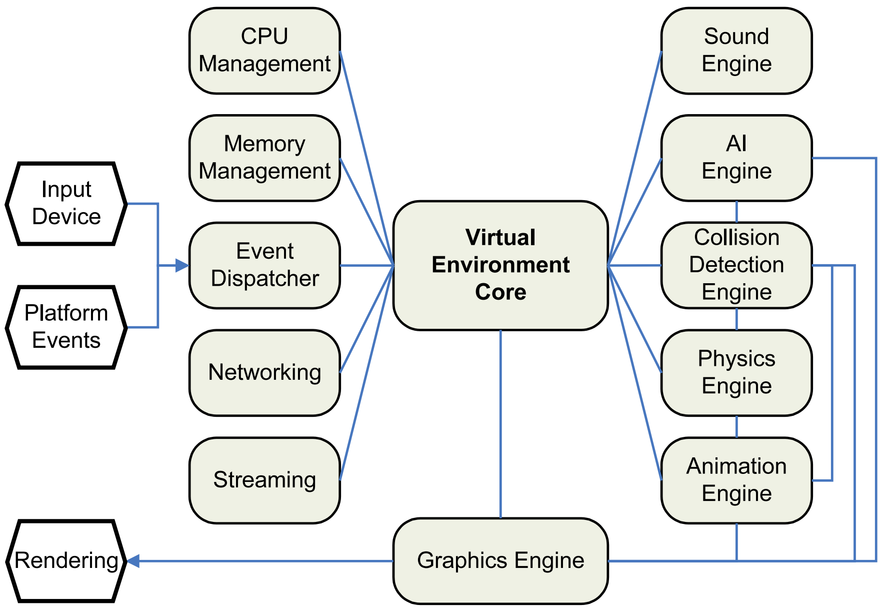


Figure 1.2: Game Software components - Modern computer games are typically made of several software components. The schematic diagram (inspired by the Quake III Arena engine (http://www.quake3arena.com/) shows how such components interact with one another. Game engines are constantly called upon by the Virtual Environment Core, according to the user input. Engines are used to load, display, and animate models; to detect collision between objects and to manage memory and CPU resources. The behaviour of virtual characters (e.g. action selection, steering and locomotion) is typically computed by the AI engine. The physics interaction of the environment content, is modeled through the physics engine. Sound engines are also common in interactive virtual environments. Music and audio effects are used to promote immersion, thus, increasing the realism of the scene.

up effective test plans. As a matter of fact, different studios have different strategies for assessing the quality of their products and none of them seems to have an optimal methodology which works for every situation [65].

1.2 Research Problem

There are many reasons why video game testing is a concrete research problem. On the one hand, the entertainment experience, virtual or otherwise, is a highly complex phenomenon involving perception and cognition [142], both considered open problems in science. In the case of realistic, interactive virtual environments, the user experience seems to be determined by a combination of psychological mindsets such as *immersion*, *flow* and *curiosity* [88]. Numerous quantitative studies have been conducted to quantify such components and determine under which conditions they may emerge or be triggered; yet, no general theory of entertainment has been formulated [1]. Without good and general models of entertainment it is hard to determine and communicate, in a non-ambiguous manner, the fun level of a game.

Along with established methods for assessing entertainment, there is also a need to find good strategies for exploring or *visiting* the game world. In modern virtual environments the user is free to move in arbitrary directions, perform a variety of actions and interact with various virtual entities in a number of different ways. This contributes to endowing the player with a sense of immersion [86]. The consequence of such realism is a dramatic game space expansion with respect to any combinatorial game such as Chess, Tic-Tac-Toe or Hex. In such games, the number of legal moves is quite limited and the game is discrete, meaning that no simultaneous move between players is allowed. Yet, the number of possible outcomes (play experiences) that can emerge from such fairly simple games can be astronomically high. The game of Chess, which provides for only 30 legal moves, enables a number of possible variations from the initial position, the so called *game-tree complexity*, of the order of 10^{120} [127]. Interactive environments are typically not discrete, they can have "boards" of arbitrarily complex 3D geometries and, differently from most combinatorial games, the legal moves (combination of legal atomic actions) they provide for are essentially infinite in number, for they are defined in a continuous time domain.

In a combinatorial game, the final position of a piece is completely determined by the initial position and the sequence of symbols or actions describing the move to perform. In a virtual environment, the final position of an object or an agent is completely determined by the initial position and the timed actions performed by the object or the agent. As such, the same move — specified as a mere sequence of actions — can position a character at an infinite number of different locations on the map (Figure 1.3). In effect, it is this enormous variety that contributes in making a game realistic in that, as we stated earlier, it lets the user freely explore the virtual world. The problem for both combinatorial games and virtual environments is that there is no guarantee that all branches of the game-tree are exciting, challenging or meaningful enough to engage the user in the playful interaction. Unfortunately, we do know yet of any general search strategy which can be followed to find such faulty branches, in case of complex virtual environments, within reasonable time.

Yet, digital games need to be verified to ensure they are at high enough quality to be released commercially. And given that no automated testing mechanism is currently available, game companies can only rely on human testing resources. Before the title is released, a number of play testers are typically hired to continuously play the game and report on the bugs they find [65]. The advantage of this solution it that it is simple and quick to employ. The disadvantages, however, are manifold. Firstly, given that the time allocated for testing is limited, it becomes necessary to determine which parts of the game-tree are convenient to explore and up to which depths. However, the optimal traversal strategy depends on what to test; thus, as long as mechanisms of automatic testing and bug detection will not become available, the effectiveness and efficiency of the traversal will always depend on the experience and skills of the game testers, producers and QA managers. Secondly, as noted earlier, it may be hard to communicate the causes of non-entertaining game-play; percepts like *mood of the scene* or *sense of realism* are hard to quantify and may be difficult to clearly report. Finally, the performance of the testing activity is strongly influenced by factors peculiar to humans, such as *frustration* and *boredom*, amongst others. Testing may become a frustrating activity, especially with early software builds when the game cannot run for more than a few minutes before crashing. Even when the testing task is clearly understood and unambiguous, making testers perform the same sequence of actions or, in general, making them play the same game over and over again, could push them to overlook defects due to the haste of getting their job done.

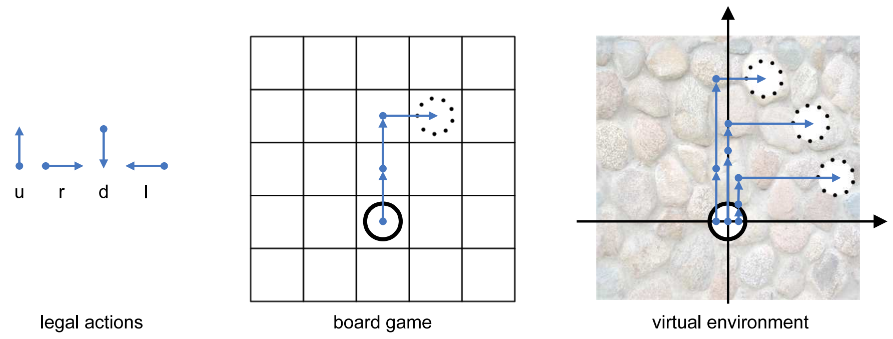


Figure 1.3: Action spaces in combinatorial games and virtual environments - A move, in a combinatorial game, is specified as a discrete sequence of actions mapping from a specific initial position to a specific final position. For example, the move $< u, u, r >$ will always move the piece two tiles forward and one tile right. By contrast, the same move, if not timed (i.e. $< (u, t_1), (u, t_2), (r, t_3) >$), can position the piece anywhere in the right-upper quadrant of the Cartesian coordinate system with origin at the current board position of the piece.

1.2.1 Towards Making the Game Testing Automatic

We believe that three major steps need to be considered in order to build any game testing mechanism. First and foremost, the type of artifacts to examine need to be clearly specified and defined; this can be anything from a flaw in the game rules (e.g. rules that enable easy wins or draws), to a flaw in the AI engine (e.g. nonsensical AI behaviours), to an instability in the physics engine (e.g. unrealistic locomotions), to a bug in the graphics engine (e.g. unrealistic or unappealing visualizations). Then, models of the correct or incorrect behaviours of the game need to be ascertained on the basis of the information available from the game software. Finally, strategies are to be found to traverse the game tree so as to emulate some human-like gameplay and expose the testing mechanism to scenarios that are likely to be experienced by human players. As we noted earlier, there is no way a highly complex game-tree can be thoroughly visited; a good option is to traverse it as a human would do. Such a strategy has proven successful in a number of cases: for measuring quality and viability of board games and facilitating the creation of new games [18]; for building mechanisms to detect unwanted AI behaviours in simple environments [32; 149] and for the on-line content creation for simple platform games [126; 137]. These can all be regarded as attempts to assess or measure quality features of a game so as to keep the player engaged in the interactive experience.

We present a general definition of virtual environments introduced by Fink et al. [40], to better describe the aforementioned steps. The authors proposed that any virtual environment can be seen as a two function system: an *update* function embodying the behaviour of the game and an *output* function, producing the output we perceive. Specifically, let S, A and O be the set of states, actions and outputs of a game respectively. With the passage of time the game takes on a sequence s_0, s_1, s_2, ..., $(s_i \in S)$ of states through the update function

$$f_{tr} : S \times A \to S \tag{1.1}$$

For each index $i \in \mathbb{N}$, $s_{i+1} = f_{tr}(s_i, a_i)$, where $a_i \in A$ is the action taken by the player between s_i and s_{i+1}. As noted by Fink, such a formulation is still valid even when the update function contains random components, provided that the seeds of the random number generators are included in the states. The output of the game can be described by another function

$$f_{out} : S \to O \tag{1.2}$$

that is while the game is in state s_i, it produces the output $o_i = f_{out}(s_i)$, yielding the observable sequence o_1, o_2, ..., $(o_i \in O)$ of outputs.

The aim of building mechanisms for the automatic testing of virtual environments can be then fulfilled once:

- we agree on the relevant behaviour of f_{tr} or f_{out} to capture or model, given the problem at hand;
- we determine what type of information is convenient to extract from S and/or O and what methods to use in order to model the target behaviour;
- we develop an effective strategy to control the update function as a human player would normally do.

In this thesis, we shall focus on the automatic detection of visual artifacts in O, caused by malfunctions in the output function f_{out}. As it will become clear in the following chapters, the information that is best suited for the automatic assessment of visual consistency will concern a small subset of S. Finally, the behavior of the *avatar* (i.e. the human-controlled character) will be captured during a normal human-play session, and replayed later for debugging purposes. This will enable us to stimulate the update function in a human-like manner.

1.2.2 Scope

Most of the current research in the area of computer game testing is focused on the qualitative and quantitative description of the player-opponent interaction; the aims of such investigations are typically fulfilled via modelling some relevant features of the update function and a *view* (a small set) of the complete states it handles [32]. The actual output perceived by the player is very often neglected. In part, this stems from the belief that entertainment can be fully captured through the symbolic information directly available from the game, with no need for audiovisual features and context [151]; in part, the choice may be justified by our limited understanding of the biological processes underpinning human perception and cognition, which enable us to make sense of the images we perceive.

In this work, we do not aim at capturing or enhancing the entertainment features of computer games. Rather, we seek to characterize and measure the consistent scene content in order to detect incorrect visualizations, that is, textures, meshes and shadowing artifacts. We postulate that anomalous visualizations can turn an entertaining interactive experience into an unpleasant or non-immersive one. Thus, the description of the visual output of the game — rather than its internal states — becomes the paramount problem of our analysis.

As we shall see, the corruption of the environment content is a concrete problem, well-known to both the developer and player sectors of the games industry. Yet, computational approaches to assist such a specific testing activity have, to our knowledge, never been investigated.

1.3 Research Questions

Under the main question of "Can the correctness of content in 3D virtual environments be automatically assessed?", we have to answer the following more specific research questions:

- Can environment anomalies be defined independently from the environment in which they appear?
- What mechanisms and descriptors are good to effectively discern between anomalous and valid visualizations?
- Will such descriptors have an object space or an image space representation?
- How can we develop a generalized approach to environment anomaly detection that treats the engine as a black box in the process of finding visual errors?

1.4 Significance

This work introduces innovative applications of machine learning and image processing research to video game and virtual environment testing. To the best of our knowledge, this possibility has never been explored. It is our belief that the solutions proposed in this work can be used to effectively assist game companies and developers of 3D applications in assessing the quality of their products. Firstly, the automatic detection of visual inconsistencies will lighten the load of inspecting the environment thoroughly and carefully to find and report as many visual artifacts as possible. Also, research conducted in this direction will stimulate the development of specific, unambiguous languages to describe artifacts that are otherwise communicated through ambiguous terminology. Finally, the use of fast estimators of environment consistency will enable a more thorough exploration of the game tree (i.e. the set of states that can be reached during a play session) for debugging purposes. The maximum number of states that can be effectively tested will depend on the hardware (e.g. consoles and computers) available during the testing phase of the product. An arbitrarily high number of visual anomaly detectors can run in parallel on the available hardware, in order to report bugs, continuously and relentlessly.

1.5 Main Contributions and Organization

This thesis makes four main contributions:

1. **A model-based method for representing image consistency:** This thesis proposes an alternative representation of the scene geometry in object space which turns out to be suitable for making predictions about the appearance of virtual entities. Based on such a representation, we propose a screen space probabilistic approach to color anomaly detection, and a combined object space geometry and color anomaly detector.

2. **A view-based approach to capture the consistent behaviour of rendering systems:** In this thesis, we develop two general connectionist approaches to color and geometry anomaly detection based on the object representation introduced by this research work. These mechanisms are able to detect mesh, color and texture corruption anomalies.

3. **An example of anomaly modelling:** This thesis introduces a new approach of shadow aliasing recognition in synthetic images based on an established image processing technique of corner detection. Despite the context-dependent nature of shadows, the context-unaware algorithm proposed exhibits efficient and accurate response to the target anomaly.

4. **Theoretical analysis of an autonomous approach to game content testing:** This thesis presents plausible approaches to the automatic extraction of testing data, from any rendering system, without significantly interfering with the 3D application to test. The proposed scheme draws from established techniques of graphics data interception and shows how such methods can be employed for automatic testing purposes.

The thesis is organized as follows. Chapter 2 presents an overview of the visual anomalies which commonly affect modern virtual environments. In this chapter, the target anomalies will be presented and an introduction to the approaches used to detect them will be given. In Chapter 3, we shall review published research in pattern recognition, relevant to this work. In particular, previous work in the areas of anomaly detection and object recognition will be presented. Next, in Chapter 4, the model based approach to visual anomaly detection will be introduced. The model based approach represents a first attempt to visual anomaly detection via reversing the graphics pipeline. More advanced visual anomaly detectors will be investigated in Chapter 5; we shall refer to such detectors as view based classifiers. The inference that view based classifiers make is based on more accurate models of color synthesis, compared to model based approaches. In Chapter 6, we will present an example of inconsistency modelling, through an appearance based detection scheme. A case study on a shadow-related anomaly will be used to show how appearance based techniques can be effectively used to detect environment-dependent anomalies. In Chapter 7, approaches to automatic retrieval of debugging data will be discussed. Throughout this document, the graphics data used for anomaly detection is assumed to be available during training and testing. The aim of this chapter is to show how such debugging information can be extracted without modifying and interfering with the game engine. Finally, conclusions will be drawn in Chapter 8, about the methods and techniques introduced, in terms of answers to the research questions posed earlier. The limitations of our approaches will be discussed, along with guidelines for future research.

2

Measuring Virtual Environment Consistency

With this chapter, we wish to give an analysis of the bugs commonly affecting computer games and 3D virtual environments in general, explaining some of their causes and consequences. This will set the stage for reasoning about the automatic detection of visual environment inconsistencies, which is the focus of this research work. In particular, in this chapter we will present the type of artifacts we want to quantify and detect, and the approach we will use to fulfill our research aims.

2.1 Unintended Artifacts in Virtual Environments

The aim of this section is twofold: to show what type of anomalies are commonly reported by the game community and therefore considered as non-negligible artifacts from the user perspective; and to briefly review the research that has been conducted in order to address them.

In order to obtain a quality sample of game bug statistics, we identified and classified game issues posted on Steam®[1], one of the most popular communication and distribution platforms[2] that gamers use to report bugs in the products they purchase. From the discussion forum of Steam, we observed that the number of issues-related posts opened every day ranged from 0 to over 60, across all games. Out of a total of

[1]http://store.steampowered.com/

[2]It has been estimated that Steam has a 70% share of the video game distribution market [50]. According to the statistics reported by the website, at the time of preparation of this document, such a platform accounts for roughly 3 million users logged in daily.

1100 game titles on Steam, we selected 100 and found that each one had at least one assiciated bug-related report. We grouped the anomalies on the basis of the aspect of the game experience that was involved and the affected components of the game. The major bug categories we have identified are:

- High Level Entertainment Issues;
- Game Usability Issues;
- Environment Inconsistencies.

2.1.1 High Level Entertainment Issues

Anomalous high level entertaining features concern bugs in the AI and physics, mechanics, play and balance of the game. Game AI and physics are the set of mechanisms embedded in the update function used to generate realistic behaviors in non-player characters (NPCs) and other virtual entities. The game *mechanics* can be defined as a set of rules intended to produce specific game experiences. This differs from *gameplay* which refers to the overall experience of playing the game. For example, the gameplay of a shooting or fighting game is to hit while not being hit. The mechanics are the processes in which such a goal is achieved, that is, attack and defense, or punch, kick, block and dodge. Finally, game *balance* describes the fairness of power in a game between multiple players or strategic options [33].

Bugs affecting such aspects of the game are caused by malfunctions of the update function and may have the effect of turning the entertaining experience into a boring, frustrating or meaningless exercise. An example of such anomalies are either nonsensical or superhuman behaviours of the AI controlling the NPCs, generating extraordinarily boring or otherwise difficult tasks to perform. Typical unrealistic AI behaviours result in NPCs engaging in meaningless exercises such as endless runs into objects and walls, or opponents being able to shoot enemies from remarkably long distances with extremely high precision and success. Bugs affecting game balance also belong to this category; in multiplayer experiences it is important that the map does not favor one team versus another. For instance, bad game balancing can be overcome by appropriately spawning weaponry and other items over the environment on the basis of the map geometry and the objectives of the game. Finally, even if the behaviour of the opponents is of the right level of challenge for the user to engage in the play experience, and the game is perfectly balanced, there is still the risk that the tasks given to the player are not particularly meaningful, in that they do not relate or fit well with the game mechanics or story progression.

In recent years, researchers have been proposing a number of qualitative and quantitative approaches to describe and quantify these issues. Perhaps, the work most closely related to ours is that of Denzinger [32], who developed evolutionary learning strategies to help human testers to detect unwanted behaviours in multi-agent systems. In particular, Denzinger assumes there are components of the system that can be selected for testing, such as the agents to test and their states (e.g. position, velocity, energy and health). From the knowledge of the task the agents are supposed to carry out (e.g. scoring a goal or rescuing other agents), Denzinger proposed to use a fitness function to evolve new testing agents, so as to optimize their performances towards their test goal. Unwanted behaviour is determined by the (manual) evaluation of the history of events generated by such agents. For example, in the case of a soccer game, the optimized agents may manage to always score a goal if performing a specific sequence of actions. Such pitfalls in the game AI are difficult to detect by a human, due to the enormous combinatorial complexity of the interactive environment. Xiao et al. [149] introduced a different strategy to semi-automated gameplay analysis. In their work they collected labeled samples from the testing system about the space of initial states and user actions. Such a *training set* was then used to learn, through decision trees, a model of the behaviour of the system. The decision of whether the learned behaviour is acceptable or not is left to the game designer. From a theoretical perspective, Aponte et al. [4] proposed a general definition of difficulty level which takes into account the user experience, assuming that the challenges of the game are clearly specified. Future developments of such work may enable quantitative evaluations of challenge and boredom in digital games.

2.1.2 Usability Issues

Usability issues arise whenever the game becomes unplayable, very hard to play or prevent the game mechanics unfolding normally. Software *crashes* and *freezes* are perhaps the worst case of usability issues, for they cause abrupt interruptions of the play experience and possible data loss. Framerate *dropping* is a less severe form of playability issue resulting in the scene being rendered at an insufficient number of frames per second. Anomalies of this kind result in a sharp motion of the camera and all animated objects in the scene; as a consequence the environment becomes difficult to control and navigate.

A rigid help system is also an example of a usability issue which may entail a poor usage of the entertaining assets featured by the application. Other usability problems commonly reported in game forums include poor customization options, wherein the user does not have direct control upon the appropriate difficulty level or game pace; obstructed views of the environment due to inappropriate camera angles; and complex command sequences to perform actions. Finally, usability bugs also include game controlling issues where the user-environment interaction becomes unpredictable due to a high *response time*, that is, the time the application takes to react to a given input action[1].

Game usability issues can be caused by faults in the game engine (i.e. the update function and its internal states), malfunctions in the game controllers or integration issues between the application and the operating system, amongst other things. To date, research investigating game usability issues is mostly qualitative, often based on questionnaires to develop heuristics that can be used to carry out usability inspection in video games. An example of such research is the work carried out by Pinelle et al. [113]. Through the analysis of 108 games from the GameSpot[2] website, the authors categorized the game anomalies found into 10 assessment heuristics. The aim of the work was to offer the possibility of evaluating the usability of games, without reviewing unnecessary technical details. In general, such a type of qualitative research aims at explaining, from a user perspective, relevant usability impediments in order to enable the design of more entertaining games.

2.1.3 Environment Inconsistencies

The third family of commonly reported bugs is what we call environment inconsistency issues. We understand consistency as the property of the virtual world to maintain the level of realism meant to be offered throughout the play experience. We use the term *level of realism* to refer to the computer graphics technique (or set of techniques) used for rendering the scene. Figure 2.1 shows examples of different levels of realism implemented in modern computer games. In order to preserve the immersive experience offered, it is important that the environment does not present the gamer with unpleasant renderings such as incorrect textures, geometries and shadows. An environment is consistent if its geometry, textures and lighting effects look correct according

[1] In game community terminology, a high response time, upon user commands, is commonly referred to as *lagging*.

[2] www.gamespot.com

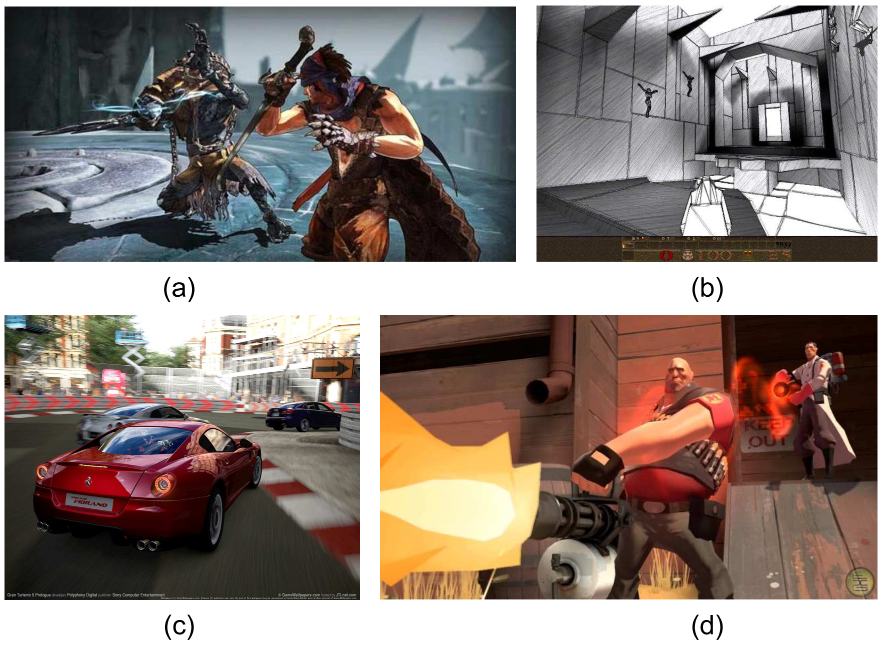


Figure 2.1: Levels of realism in modern computer games - Today, game environments can be rendered using many different styles. Some examples are: graphic novel (a), pencil sketch (b), photo-realistic (c) and cartoon-like (d) fashion.

to the level of realism and meaning the entities acquire in the virtual world. In the following sections, we will discuss some of the most common environment inconsistencies reported by the game players. These are incorrect texturing, shadowing, geometry and collision detection.

Incorrect texturing/coloring

Texturing is a critical component of virtual environments, for it is a fast and remarkably effective way of deceiving the eyes of the user into perceiving materials rather than just plain polygons [21]. Such an illusion, however, breaks in case of *distortion* and *corruption* by which the mapping of the texture on the mesh typically results in overly elongated stripes of uniform color, or in the texture being acciden-

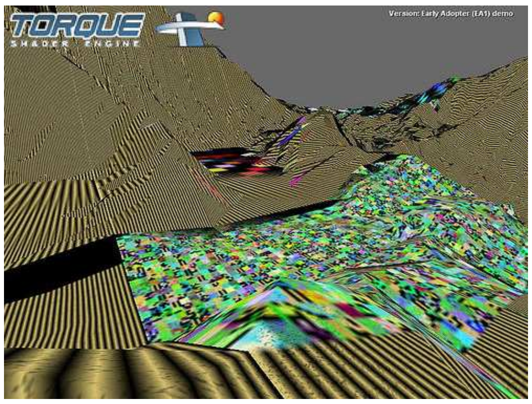


Figure 2.2: Texture Anomaly

tally replaced with some other information from the video memory. In theory, any method that relies on texture mechanisms (e.g. *bump mapping* [12] and *displacement mapping* [132]) may undergo texture issues, resulting in unrealistic renderings.

Incorrect shadowing

Shadowing is an important aspect of realism as shadows provide vital visual clues that communicate spatial relationships and information to the viewer. It is therefore not surprising for players to be particularly pedantic on the correct, or at least plausible, visualization of shadows in the virtual environment they are exploring. Since rendering highly accurate shadows in real-time demands high computing power, a number of fast approaches to shadowing have been proposed over the years. For a survey on real-time shadow algorithms, the reader is referred to [56] and [124]. Each one of these techniques has some known issues that, if neglected by the designer, may lead to visually unappealing renderings. Common shadow anomaly reports include *aliasing effects* such as jagged edges and Moiré patterns, missing shadows and shadows cast onto invisible geometries.

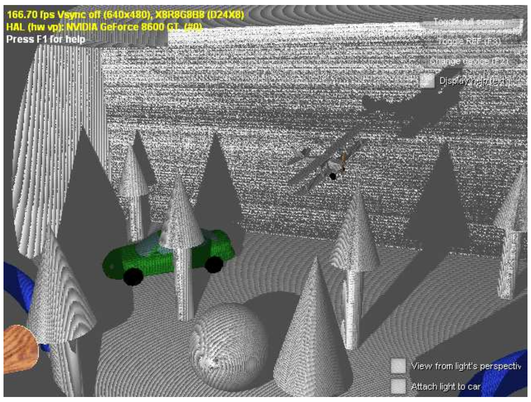

166.70 fps Vsync off (640x480), X8R8G8B8 (D24X8)
HAL (hw vp): NVIDIA GeForce 8600 GT (#0)
Press F1 for help
View from light's perspectiv
Attach light to car

Figure 2.3: Shadow Anomaly

Incorrect object geometry

Virtual environments are visually defined by the geometric entities populating them. Such entities represent models of real or fictional objects with which the user interacts and by which she carries out her own tasks. From the player perspective, the objects may present an awkward shape according to their texture or they may be displayed in improper locations and poses, given their purpose and the context in which

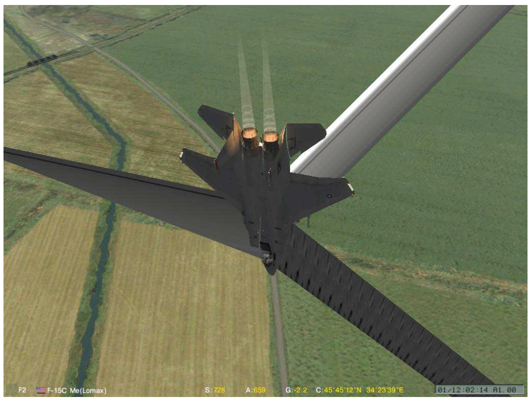


Figure 2.4: Geometry Anomaly

they appear. Such artifacts, as we shall see shortly, are typically the by-product of software bugs (at the application and/or driver level) or hardware malfunctions.

Incorrect bounding volumes and collision detection

Collision detection is the mechanism by which virtual environments emulate the solidity property of objects, typically, through physics engines. To give the illusion of solidity, *bounding volumes* (e.g. boxes or spheres) are used. These are meshes circumscribing the target object and representing regions within which the rest of the geometries are not allowed to enter. If the mechanism is not properly implemented, objects will clip into each other, producing visually unappealing artifacts. Other collision issues concern the interaction between the camera and the environment; for example, improper bounding volumes may allow the camera to travel and see through walls.

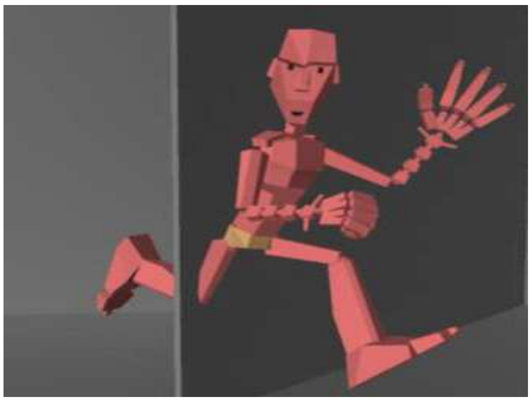


Figure 2.5: Collision Detection Anomaly

2.2 Targeted Environment Inconsistencies

As stated at the beginning of this chapter, the aim of this research is to address environment inconsistencies. In particular, in this work the inconsistencies we will target are *geometry corruption*, *color corruption* and *shadow aliasing*. This type of anomalies will be presented in the following sections. A brief background on the relevant terms and concepts of current real-time computer graphics technology will also be given, in order to reinforce understanding of the virtual environment components affected by the aforementioned anomalies.

2.2.1 Geometry Representation and Mesh Corruption

In real-time computer graphics, the most common representation of a virtual geometry is the *polygonal mesh* or simply mesh [43]. A mesh is a set of connected polygonally bounded planar surfaces made of vertices and polygons connected such that each edge is shared by at most two polygons (Figure 2.6). A vertex is a point where two edges of a polygon meet; an edge, in a polygonal mesh connects two vertices and a vertex is shared by at least two edges. Three-dimensional vertices are defined through position vectors, but they can also be augmented by other information such as color, normal and tangent vectors. The position of a vertex is expressed through an (x, y, z, w) vector, were w is a homogeneous component, included in order to express any geometric transformation through matrix multiplication [58].

The main advantage of a mesh lies in its capability in linearly approximating any geometry with arbitrary accuracy [14]. In general, the greater the vertex density of the mesh, the better the approximation of the equivalent surface, but also the more computing power is needed for its real-time rendering. The set of meshes composing a virtual object is typically stored into a file called a *model file* [43] so as to be easily imported and used by any 3D application. Color and textures associated with the object are typically stored in the model file.

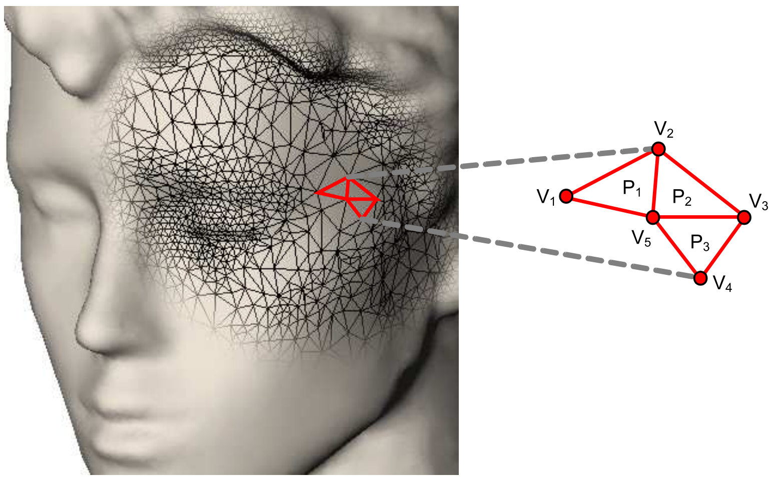


Figure 2.6: Polygon mesh example - Meshes are sets of vertices (V) and polygons (P) connected such that each edge between two vertices is shared by at most two polygons.

Geometry or mesh corruption is a common visual artifacts, which we ascribe to the family of environment inconsistencies. Corrupted meshes can ultimately be attributed to anomalous video memory (or video RAM) contents, caused by either software or hardware malfunctions, or both. The video memory holds, amongst other things, the position, normals and tangents of vertices that make up the object meshes. This information is stored in *vertex buffers*, a sequence of contiguous video memory locations that contains vertex data. Corrupted information in the memory buffers can therefore result in the vertex properties being changed (i.e. translated, rotated and/or scaled). Common geometry issues result in *spike* patterns (Figure 2.4); these are the result of vertices being displaced away from from the intended position on screen. As for the vertex position, normals, tangents and indices[1] can too become corrupted. The visual implications of such malfunctions in the final images are very hard to predict. Geometry anomalies of severe magnitude, in general, result in severe occlusions and clutter, thus preventing the user correctly perceiving, and therefore interacting, with the 3D application.

2.2.2 Color Synthesis and Corruption

A large number of environment inconsistencies fall into the broad category of color corruption. The color of virtual objects can is typically represented through textures and/or color vectors, to be mapped to the underpinning mesh. Textures are 2D matrices of pixels addressed through two-dimensional coordinate systems. Such coordinates are used to map the texture onto the geometry, thereby significantly increasing the realism and detail of the object (Figure 2.7).

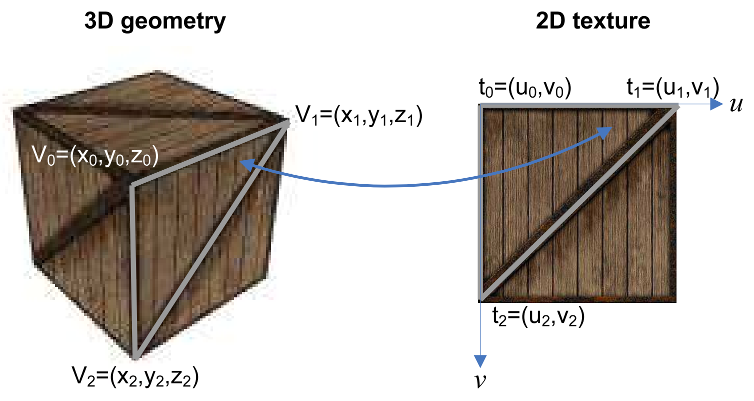


Figure 2.7: Texture coordinates mapping - The coordinates (u, v) identify the element or *texel* on the texture image which will be sampled according to the pixel position. The (x_i, y_i, z_i) vertex coordinates are mapped, by the application, to the (u_i, v_i) coordinates. All pixels lying within the triangle $V_0\widehat{V_1}V_2$ are textured by using the triangular texture patch $t_0\widehat{t_1}t_2$.

Colors can also be assigned to each vertex of the object mesh, via (R, G, B, α) vectors representing the red, green, blue and *alpha* channels respectively; where alpha is the opacity percentage (transparency) of the vertex. The final color of the object pixels is then computed by the GPU via interpolating the color values from nearby vertices.

The color distribution over the objects surface, in the final image, varies according to the lighting of the scene. In computer graphics, light models are typically computed

[1]Index buffers indicate the order in which the related vertices in the vertex buffer are to be assembled by the GPU.

by additive mixtures of diffuse, ambient and specular contributions [90]. Lights can be represented by points (e.g spotlights) or surfaces. In the latter case, each point within the surface is a light source. Finally, lights are also used to add shadows and global illumination effects such as caustics [87], radiosity [80] and bloom [110].

In order to increase surface details of the objects, and therefore the photo-realism of the scene, *displacement mapping* [132] and *bump mapping* [12] techniques are often used. The former, consist of changing the original mesh by displacing vertices and/or pixels along their normals, according to some *height map* (a 2D texture). Similarly, bump mapping techniques perturb vertex normals during lighting calculations (the actual geometry is not modified), simulating small protrusions or 'bumps' on the surface.

Color vectors, textures, height and bump maps are all stored in the video memory of the rendering system. The video memory also holds an area called *frame buffer*, a linear array containing the image that has been, or will be, rendered (i.e. the red, green and blue color intensities of the final pixels on screen). Color corruption or inconsistency occurs when the color content of the video memory becomes corrupted. Accordingly, the anomalous content of texture, color and frame buffers can all result in visually displeasing artifacts (see Figure 2.2 for an example).

From a testing perspective, it would be desirable to have mechanisms to detect whether the color and shape of virtual objects has been properly rendered. Our aim is to investigate how color and geometry corruption can be described and modeled by extracting the content of the video memory, as the scene is synthesized by the GPU. The main advantage of performing this type of analysis is that colors and geometries can be defined and extracted in an environment or application-independent manner. Such an extraction process is facilitated by the use of standard graphics libraries with enforced data formats.

2.2.3 Shadow Aliasing

As stated earlier, shadows contribute to the realistic appearance of virtual environments. In order for shadows to endow the player with a sense of realism, it is important that such artifacts themselves appear realistic. A critical step in the implementation of numerous game assets is the tuning of various software parameters to produce the intended appearance under a large enough number of environment configurations. This is the case of many texturing, shadowing and lighting techniques, among others. For those effects whose output does not depend on the entire content of the scene, the appropriate parameters can be easily determined by heuristics concerning the object geometry and its position with respect to the camera only. As an example, consider the *texture mipmapping* method used for mapping textures to different triangle sizes on screen [90]. In the mipmapping algorithm, the original texture is resized to smaller, filtered copies to be used for rendering distant triangles. The right texture level or resolution, with respect to the triangle size on screen, is typically decided by the algorithm proposed by Williams [145], which guarantees smooth, relatively imperceptible transitions between different resolutions.

For other visual effects, such as shadowing, the final output depends on scene content rather than on the individual entities. As such it is hard to find methods which guarantee that no artifacts will ever affect the rendering. Simply put, the more environmental influence on synthesizing the final image, the more variables will need to be considered for a bug-free rendering. As a consequence, such effects and their tuning parameters need to be tailored to the specific environment, at testing time. An improper tuning may lead to unrealistic appearances like the ones depicted in Figure 2.8.

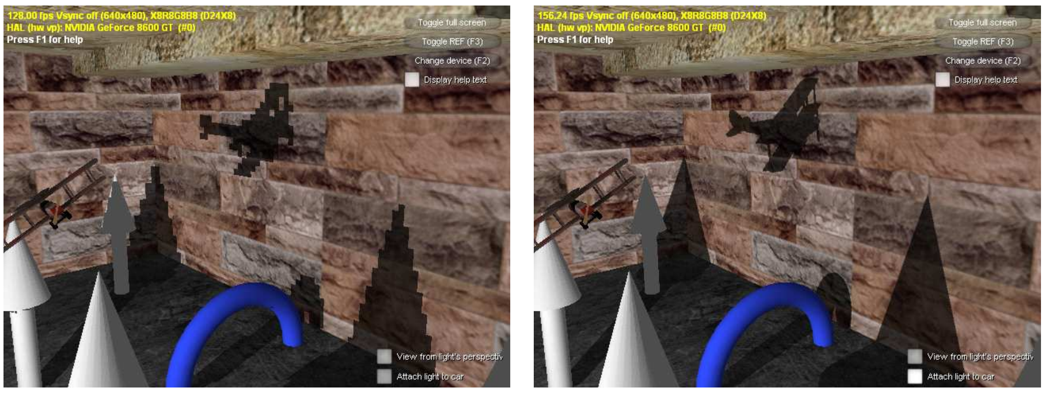

128.00 fps Vsync off (640x480), X8R8G8B8 (D24X8)
HAL (hw vp): NVIDIA GeForce 8600 GT (#0)
Press F1 for help
Toggle full screen
Toggle REF (F3)
Change device (F2)
Display help text
View from light's perspectiv
Attach light to car
156.24 fps Vsync off (640x480), X8R8G8B8 (D24X8)
HAL (hw vp): NVIDIA GeForce 8600 GT (#0)
Press F1 for help
Toggle full screen
Toggle REF (F3)
Change device (F2)
Display help text
View from light's perspectiv
Attach light to car
(a)
(b)

Figure 2.8: Example of shadow aliasing - Shadowing algorithms need to be tuned according to the environment so as to avoid visual artifacts. The pictures shows an example of a shadow aliasing anomaly (a), resulting in blocky edges of the shadow silhouette, with respect to a realistic shadow (b).

The shadow mapping algorithm, at the time of this work, represents one of the most commonly employed techniques for generating realistic shadows in interactive 3D environments [125]. Numerous improvements and modifications to the original shadow mapping algorithm have been proposed to reduce the visual artifacts peculiar to the original version of the algorithm proposed by Williams [144]. These improvements generally produce more realistic visualizations, but they entail more complex algorithms and extra parameters to tune.

From a testing perspective, it is interesting to determine whether the final image contains shadow aliasing artifacts, independently from the complexity of the scene or of the underpinning algorithm. As with color and geometry issues, shadows will be assessed by extracting and analyzing the related data from the video memory of the rendering system. As discussed earlier, this approach enables the construction of environment-independent, effective anomaly detection systems.

2.2.4 On Game States, Functions and Rendering Systems

Under the formal definition introduced in Chapter 1, any virtual environment can be seen as a two-function system. These two functions being the update function, f_{tr} (Equation 1.1), and the output function f_{out} (Equation 1.2). This latter, operates on the game states in S and produces the sequence of images in O that the user perceives. In fact, the output function can be further decomposed into three modules: the one producing the visual output, the one producing the acoustic output, and the one producing the *haptic* (or tactile) feedback (e.g. forces and vibrations) to the user, given the previous game states. In this work, we are only interested in studying the visual correctness of virtual entities; thus, we shall use the symbol f_{out} as to indicate the visual output module only. We will use the terms *output function* and *rendering system* interchangeably, in order to indicate the overall process of image synthesis.

The game states in S can be thought of as the content of the (CPU and video) memory addressed by the 3D application. However, the subset of S, input to f_{out}, is the graphics data used by the pipeline to render the scene. This includes vertex and index buffers, textures, transformation matrices and light sources. The set O, is the set of frames generated by the game, that is, the content of the frame buffer. The function f_{out} is partially implemented via software and partially via hardware. The software part corresponds to the graphics interfaces used by the application, such as DirectX and OpenGL; the hardware side is the GPU of the machine in which the application runs. Because a game is typically expected to run on a number of different GPUs, f_{out} is typically implemented in many different ways. The graphics code need to be tailored to the GPU on which the 3D application is expected to run, because different GPUs feature different rendering capabilities. Thus, given that f_{out} may ultimately denote a collection of implemented graphics algorithms, the same input state S may results in different outputs O: unrealistic renderings for poorly performing machines, and highly realistic scenarios for highly performing hardware.

2.3 Appearance Based versus Behaviour Based Detection

The main focus of this work is the detection of visual anomalies via the analysis of the images in O and other graphics data in S contained in the video memory of the rendering system. To fulfill our aim, we will investigate various techniques from the field of computer vision and machine learning. In particular, through image processing mechanisms, we aim at describing the appearance of objects and effects. Then, when necessary, we shall employ pattern recognition algorithms to model the consistent behaviour of the rendering system and discriminate between anomalous and correct visualizations. The rationale behind such a strategy is to maintain the amount of information required from the game (e.g. vertices, textures, lights and graphics code) to a minimum, and therefore make the solution as general as possible. The amount and type of data needed from the game to build robust classifiers is indeed one of our research questions (see Chapter 1). Specifically, the problem we will be facing is the following:

> *Given the anomaly we wish to target, what type of information is good to extract from the application to test, and how do we process it so as to effectively discern between anomalous and valid visualizations?*

A characteristic common to both objects and related anomalies is their ever changing appearance. As for real objects, 3D virtual geometries are subjected to camera state changes producing geometric transformations which result in changes in the two dimensional shape we perceive. The color of the object may change too, according to the light models implemented in the environment, and the properties of the surfaces of the objects. An incorrect object (texture or shape) changes as much as a correct one does, as both correct and anomalous entities are likely to undergo the same rendering processes. The challenge is to correctly discriminate an ever changing correct rendering from an ever changing artifact.

One way to look at this problem is through an *appearance based* pattern recognition scheme, considering the anomaly as a target object or event to recognize. To this end, one can look for relevant properties or features in the image at hand, that are characteristic to the artifact and possibly to nothing else in the scene. We use this approach in Chapter 6 for detecting shadow mapping aliasing. In our method, we make use of a computer vision technique to extract features characteristic to the target artifact. As we shall see, such features can be extracted from images that have been preprocessed so as to remove irrelevant information and to make the system more robust (Figure 2.9).

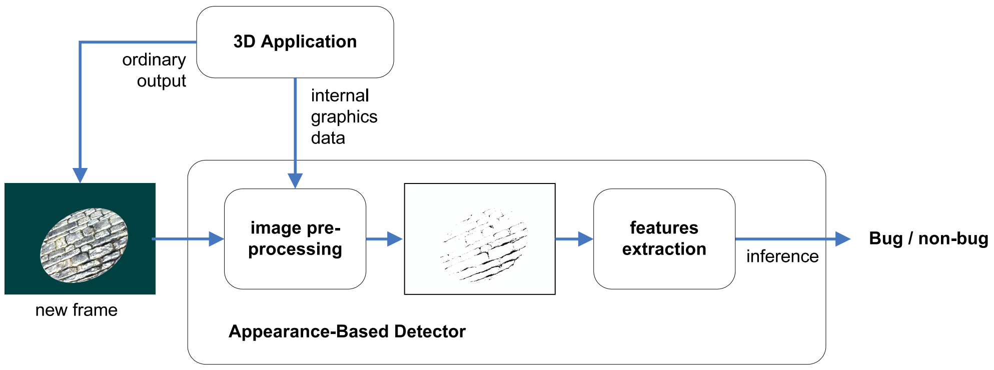


Figure 2.9: Appearance based scheme - The approach aims at detecting patterns peculiar to the target anomaly. To make the system more robust, the target image is first pre-processed to remove irrelevant information. Then, characteristic features are extracted to determine whether the anomaly is present or not. In the appearance based scheme the *a priori* knowledge of the anomaly is embedded in the classifier.

A different approach to anomaly detection can be based on the knowledge of the correct behaviour of the rendering system. Such a behaviour can be either encoded by the designer of the detector, or learned by the detector itself. This problem is very similar to the novelty detection problem, for which a number of techniques exist. These approaches are reviewed in the next chapter and some of them used in this research work. As for any ordinary novelty detection problem, we are faced with the challenge that we can never train a learning system on all possible events (visualizations) that the target environment can produce. Nevertheless, it becomes important to differentiate between normal and anomalous information during testing, even for samples that have never been observed, such as objects seen from different camera angles.

To address such a camera-related issue we will follow two different approaches: a *model based* method and a *view based* method. Following the model based scheme (Chapter 4), we will aim at modelling the visual data about the objects on the basis of its geometric and color properties and use this information to estimate whether a test scene is consistent or not. To that end, we treat geometry and color separately. Geometry will be assessed in a canonical space, that is, the 3D local space in which the objects are defined. Color will be assessed in another metric space (the HSV space) through color histograms. We will also introduce a statistical approach to measure color consistency to predicting consistent visualizations on the basis of the statistical color properties of the objects. The probability estimate will be used to denote visually novel (inconsistent) events.

The view based approach we will explore is based on connectionist technology. We propose two different architectures for our problem; these are *neural-networks* and *self-organizing map* based architectures. In the former case, we train a neural network to find a good mapping between the internal information available from the game about the individual object to render, and the visual appearance of the object itself. The detection of the anomaly will then be based on thresholding the Euclidean distance between the output from the trained network and the target visualization. High distance values will indicate novel, or otherwise inconsistent, samples. In the self-organizing-map approach, the state of the object is combined with the description of the object appearance in order to learn the distribution of the consistent visualization in an unsupervised and robust manner. After training, the detection of the anomaly will be based on the Euclidean distance computed between the target sample and the closest unit (neuron) to it. Similar to the neural-network approach, high distance values will indicate inconsistent renderings. In all novelty detection approaches we present, the training data is some information from the game (e.g. images and internal variables) that is assumed to be correct or consistent. The general architecture of the behaviour-based scheme is depicted in Figure 2.10.

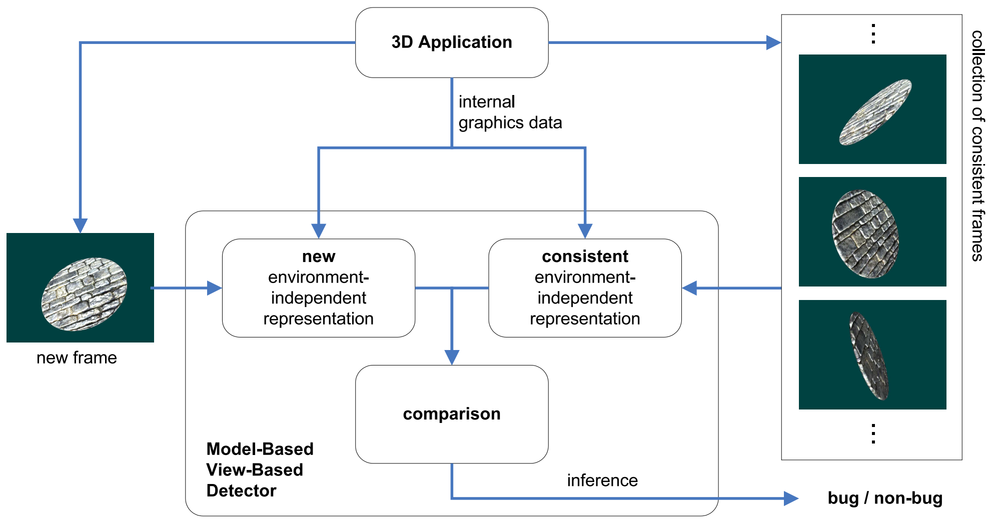

3D Application
internal graphics data
collection of consistent frames
new frame
new environment-independent representation
consistent environment-independent representation
comparison
Model-Based View-Based Detector
inference
bug / non-bug

Figure 2.10: Model Based and View Based scheme - Both model based and view based approaches aim at modelling some behaviour of a bug-free rendering system to predict what the consistent appearance is going to be in new images and graphics data. The process involves two stages: a training phase and a test phase. During training, the target behaviour is modeled from a set of consistent images and internal graphics data. During testing the new appearance is predicted using new graphics data and compared to the actual one for bug detection.

All novelty detection strategies presented in this work are based on the assumption that the only information about the game states disclosed to the testing system is the geometric transformation operations used by the rendering system to render the objects and *matrix maps*, that is, images which enable the classifier to uniquely identify the geometric transformations applied to visible objects in the scene. This choice of extracting only a limited amount of data is motivated by two main reasons. On the one hand, we wish to treat the game engine as a black box, so as to minimally interfere with the normal process of image synthesis and introduce in this way additional artifacts to the final rendering. The black box assumption, by providing the appropriate testing interfaces, will also reduce the effort from the developers of the environment in making their application compliant to our testing system[1]. On the other hand, the more information we retrieve from the application, the higher becomes the complexity of the detectors used for classifying the output of the rendering system. Furthermore, as the number of variables to consider increases, the number of minimum training samples required for learning increases too. As Markou and Singh noted [94], the amount of training data required for a neural network, as the input dimensionality increases, may increase as a power of the number of dimensions. As we shall see in Chapter 4, in order to learn the consistent behaviour of the output function, some consistent information needs to be available for training the detectors. Such information, to be considered consistent, needs to be validated by a human. Hence, smaller amounts of training data will make our solution more feasible due to less time spent visually validating the training set.

The novelty detection algorithms learn the consistent appearance of individual objects based only on the information from the game related to the objects themselves. This implies the assumption that the information from the game not directly related to the target object will not affect the appearance of the object itself. Moreover, we will assume that the states previous to the current one do not influence the current rendering. Clearly, all these assumptions do not hold true in general. For example, the target object may be occluded by the content of the scene; an occlusion involves an object shape on screen which cannot be described solely on the state (e.g. position and orientation in the 3D world) of the object itself. Also, for highly realistic environments, the color of the target object may depend on the color and position of the surrounding geometries; this is, for example, the case of specular and refractive surfaces of the objects and *light occluding* geometry generating shadows. Finally, the game may feature

[1]Techniques of automatic, environment-independent graphics data extraction will be discussed later in Chapter 7.

visual effects — such as *motion blur* — whose output depends on both the current and the previous state of the game — for example the previous position and current velocity of the objects. As we shall see, however, environment influence in the data set (i.e. set of images from the game) can be treated as noise. If this noise is mild and the detectors do not overfit the data (i.e. do not learn the noise), robust inference can be made even in the case of photo-realistic environments.

2.4 Experimental Environments and System Validation

Our novelty detectors have been calibrated and validated on a freely available game environment which we modified to generate the information needed by our algorithms and to reproduce the artifacts we wanted to target. Such an environment is the *Microsoft Racing Game*, a close to professional quality game developed by Microsoft as a *starter kit* for the Microsoft® Xna®[1] development framework. This environment featured several objects (more than 70) and two different levels of realism. In particular, the basic level of realism (Figure 2.11 (a)) featured displacement mapping, environment mapping and shadow mapping, amongst other effects. The advanced level of realism included post-process *blooming* and *blurring* (Figure 2.11 (b)). We modified the engine so as to include a third rendering approach. This was the cartoon-like cel shading, which endowed the virtual entities with a cartoon-like appearance (Figure 2.11 (c)). As we shall see in later chapters, the validation of all model based and view based algorithms is performed upon three different objects of different geometric complexity. Also, color tests were carried out across the three levels of realism presented earlier, so as to measure the effectiveness of our systems under different lighting conditions.
The shadow aliasing detector, instead, was validated on a Microsoft® DirectX® SDK virtual environment called *ShadowMap* (Figure 2.12). Such an environment implemented the standard version of the shadow mapping algorithm [144].

2.5 Summary

In this chapter we have presented a broad overview of the anomalies that most commonly affect computer games and virtual environments in general. We have shown that, although quantitative and qualitative approaches have been proposed to address high level entertainment and usability issues, no contribution has been made towards easing

[1]http://creators.xna.com/en-US/education/starterkits/

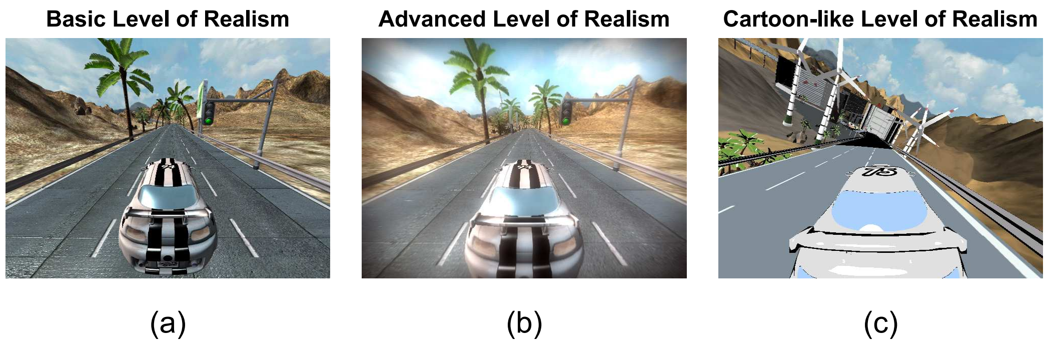


Figure 2.11: Microsoft Racing Game - To test our system, we used the Microsoft Racing Game. Besides the two levels of realism implemented by the standard version of the game — i.e. basic (a) and advanced graphics effects (b) — then extended the game engine and implemented a cartoon-like version of it (c).

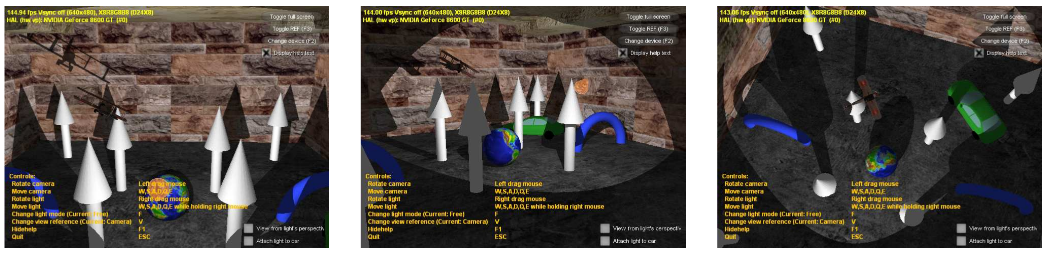


Figure 2.12: Shadow Mapping Sample Environment - Our shadow aliasing detector was validated on a virtual environment sample bundled with the Microsoft DirectX 10 SDK, called ShadowMap.

the manual testing of the environment integrity. In general, drawing a clear boundary between what is visually wrong or unpleasant and what is not is a fairly hard task. Ultimately, the perception of correctness may change depending on what is expected from, or known about, the game. The level of realism of the scene is a good example of such a phenomena; a cartoon-like visualization of an object should not be considered anomalous if the game is entirely rendered with such a technique. In contrast, if an object is rendered in some cartoon-like fashion within a photo-realistic scene, that object is likely to be perceived as anomalous, unless the peculiar appearance serves a specific purpose or, simply, looks more visually appealing that way. In the following chapters, we will develop methods whose effectiveness can be calibrated to the level or realism featured by the environment. We will target issues such as color and geometry corruption and shadow aliasing.

Assessing virtual environments for defect detection is ultimately a classification problem which, as we shall see, can be addressed through what we have named appearance-based methods, aiming at detecting patterns peculiar to the target anomaly; and model and view based approaches, aiming at modelling some behaviour of the rendering system.

3

Related Work on Pattern Recognition

In Chapter 2, we reviewed quantitative and qualitative approaches to autonomous game testing. The purpose of this chapter is to present relevant work that has been carried out in the domain of pattern recognition. The task of recognizing patterns in images is central to this research work, for the detection of anomalous visual events ultimately is a pattern recognition problem. In our case, the patterns of interest are those that significantly differ from the patterns emerging from the normal behaviour of the rendering system. Specifically, in this chapter we shall review:

- current methods to novelty and anomaly detection, in image processing and machine learning research;
- current mechanisms of object recognition, in computer vision research.

3.1 The Pattern Recognition Problem

Pattern recognition is a scientific discipline, covering developments in the areas of statistics and machine learning, computer vision, psychology and physiology, amongst others [135]. The ultimate goal of all pattern recognition problems is to classify observed events into a number of categories or classes. The nature of the target events can be arbitrary, ranging from acoustic signals, to images, to sequences of words or letters, to DNA strands. More formally, a *pattern* is often defined as a p-dimensional data vector $\mathbf{x} = (x_1, x_2, \ldots, x_p)^T$ of features (or attributes) of the event of interest. Examples of such attributes are Discrete Cosine Transform (DCT) coefficients from acoustic waveforms [156], raw pixels from an image [148] or nucleotides from a DNA sequence [116]. Associated with each pattern there may be a categorical variable z, denoting the class membership of the feature vector. Assuming there exist C classes, if $z = i$ then the related pattern belongs to the i^{th} class, $i \in \{1, \dots, C\}$. The set of patterns of known classes is typically denoted by $\{(\mathbf{x}_i, z_i) \mid i = 1, 2, \dots, n\}$, where n is the number of classified patterns. This set is often referred to as *training set*[1]. The task of building a pattern recognizer consists in building a model that allows a mapping from the feature space of $\mathbf{x}$ to the set of classes, for any unknown pattern. If the class labels are not available, patterns are grouped into classes according to some similarity measure. The general stages involved in building a pattern classifier are depicted in Figure 3.1. These are *sensing*, *feature extraction/selection*, *modelling* and *validation*.

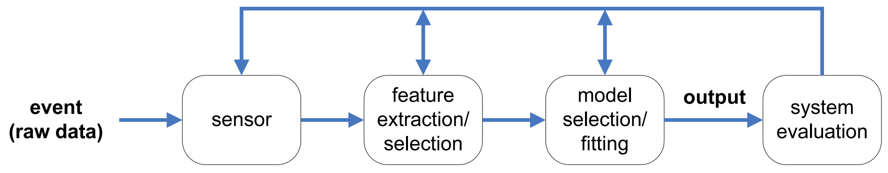


Figure 3.1: Pattern Recognition design stages - Stages involved in the design of a pattern recognition system. Illustration inspired by Theodoridis and Koutroumbas [135]

A sensor is a device (e.g. a microphone, a camera or a thermometer) that converts real world phenomena into analog and/or digital signals (e.g. a sequence of bits). Clearly, if the information is already available in a digital format, the sensor stage will not be present. The feature extraction and selection is critical to any pattern recognition task. This stage includes all processes involved in reducing the complexity (i.e. dimensionality) of the incoming row data, removing redundant information and transforming it into a more appropriate form for the problem of interest. For example, in some acoustic and visual pattern recognition tasks, the input signal is projected onto orthonormal bases, via using Discrete Cosine Transform, Principal Component Analysis and Wavelet Decomposition [115]. Next, a model is *trained* in order to build the mapping between the feature vectors and the classes to which they belong. Learning can be carried out in a *supervised* or *unsupervised* manner, depending on the availability of the class labels in the training set. Finally, the output of the classifier is evaluated and the system parameters adjusted or changed in order for the predicted value to be as close as possible to the true value, in some optimal sense [135]. In the following sections, we will discuss two areas of pattern recognition relevant to our research work,

[1]The training set may or may not contain the class labels z_i.

that is, anomaly detection and object recognition. However, before doing so we need to point out the issues that are common to the design of any classifier.

Firstly, if a classifier is too complex (i.e. it has too many parameters to tune) it may *overfit* the data, that is, learn random errors or noise. Conversely, if the model is too simple, it may fail to represent complex structures of the data set (*underfitting*). An example of overfitting is given by a polynomial curve used to fit a set of points. If the degree of the polynomial is too high, the curve will fit the data set "too well" (in the fitting error sense), by modelling every fluctuation of the data, possibly due to noise. On the other hand, if the degree of the polynomial is too low, the fitting error will be large and the underlying variability of the data will not be captured. Choosing the appropriate model is an exercise in *model selection.* In general, high dimensional input spaces require complex models in order to fit the data. This is why, the input complexity is often reduced via appropriate feature selection. The difficulty in selecting the appropriate model comes also from the fact that it is hard to determine what is structure and what is noise in the data.

Secondly, the performance of a particular classifier on a given data set depends on both the classifier and the data [139]. Typically, trial-and-error processes are required to find the model that best describes a given data set.

Finally, even the best fitting model for the given training set may perform badly on the test set, that is, the set of unobserved events. The reason being, the training and test data may be significantly different. For example, the test data may be subject to more noise, with respect to the training data. These are all factors that should be accounted for in classifier design.

3.2 Novelty and Anomaly Detection

Our research work aims at detecting defects in realistic images generated by 3D applications. Assessing correctness is similar to the general problem of novelty or anomaly detection for which several approaches already exist. In image processing and machine learning research, anomaly detection refers to the problem of finding patterns in data that are new or unknown, that is, not conforming to some well defined notion of normal behaviour [23]. Anomalies in the data may translate to significant events in the real world. For instance, novel traffic patterns in a computer network may result from attacks or intrusions [27]; anomalous electroencephalogram (EEG) scans may indicate seizures [46]; anomalous readings from space craft sensors could signify defects in some component of the space craft [45]. The problem of detecting novelty is an extremely challenging classification task [94]. Typically, detection of novelty is performed by defining regions representing normal behaviour and ascribing "interestingness" to the data not belonging to such regions. However, defining regions encompassing every possible normal behaviour may be extremely difficult, especially when the data is dynamic (e.g. audio and video signals) and the samples available are limited in number. Moreover, the notion of anomaly differs across application domains, and so does the definition of noise which typically contaminates the data. For these reasons, there is no single optimal model for detecting anomalies. The success of the detection depends on both the technique used and the statistical properties of the observations.

Anomalies are typically classified into two main categories, these are *point* anomalies and *contextual* anomalies. If the anomaly can be defined independently from the context in which it appears, then it is referred to as a point anomaly . Otherwise, if the data instance is anomalous in some contexts but not otherwise, then the related anomaly is called a contextual anomaly [24]. One example of a point anomaly is a very high (or low) expenditure, among the average expenditures in credit card transactions. By contrast, the abnormal behaviour of the time-varying temperature of an area is an example of contextual anomaly. Here, the contextual attribute is time, whereas the non-contextual attribute (observation) is the temperature. Any point anomaly can be turned into a contextual anomaly by incorporating context information into the data.

In detection tasks, the data used for generating the model is generally associated with labels denoting whether the instance is anomalous (novel) or not. Labeling is often a tedious process, for it requires the effort of human experts who need to carefully inspect the training set and tag what is perceived as anomalous or otherwise. In some cases, labeling is carried out only for positive samples, as a comprehensive set of anomalous events may be hard to collect and hence the anomalous set would be considerably smaller than the set of normal events [23]. Based on what type of labels are available, anomaly detectors can operate in a *supervised*, *semi-supervised* or *unsupervised* manner. Supervised anomaly detectors require that both positive and negative labels are available denoting known and anomalous events respectively. A typical supervised anomaly detector consists of a predictive model built on a set of labeled (normal and anomalous) samples. The supervised anomaly detector then clusters new observations into normal versus anomalous classes. In the semi-supervised scheme, labels are assumed to be available only for the normal class. Because semi-supervised approaches do not require labels for the anomaly class, they are typically more appealing than supervised techniques. The semi-supervised classifier will ultimately be a model of the normal behaviour. Bad predictions from the model will be ascribed to anomalous observations. Finally, unsupervised techniques do not require labels as these methods rely on unsupervised clustering techniques [84]. These methods first try to find anomalies "buried" in the unlabeled data set, in which the normal instances are assumed to be far more frequent than the anomalies. Upon the labels produced by the unsupervised mechanism, ordinary supervised anomaly detectors are than used. Figure 3.2 depicts a simple two-dimensional example of normal versus anomalous events and related labels.

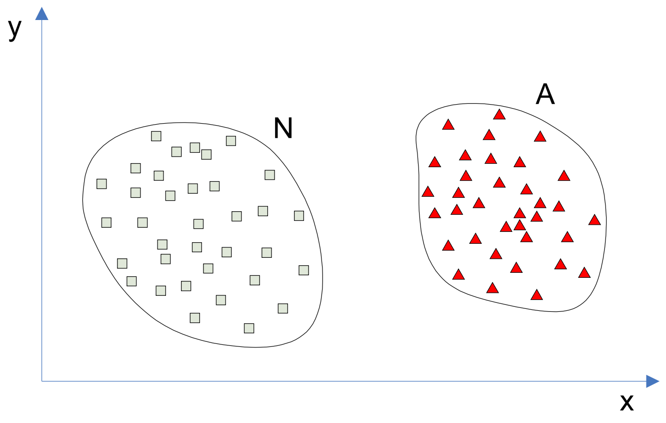


Figure 3.2: A simple example of anomalies - Anomalies are patterns in data that do not conform to a well defined notion of normal behaviour. If the set of normal (N) and anomalous (A) events is labeled, supervised techniques can be used to learn the boundaries of the two sets. In some other cases, only the set of N normal events is known a priori. In such cases, the anomaly detection task can be accomplished by using semi-supervised learning techniques. Finally, if neither N nor A is labeled, unsupervised approaches are used in order to determine which events are most frequent (therefore, supposedly normal) in the data set.

Besides architecture and type of anomaly handled, detectors are also defined by the way the anomalies are reported. In particular, the output of a detector can be a *score* value, indicating the level of novelty detected or a binary *label*, indicating whether the instance belongs to the normal class or not.

3.3 Anomaly Detection Techniques

Novelty detection approaches fall into five major categories. These are *classification-based*, *nearest neighbour-based*, *clustering-based*, *statistical-based* and *information theoretic-based* approaches [23]. Such techniques will be reviewed in the following sections.

3.3.1 Classification Based Approaches

Classification based anomaly detection operates under the assumption that normal and anomalous classes in the given feature space can be discriminated, via some classification technique. Depending on the detection problem and on the availability of training labels, classification based approaches can be further grouped into *one-class* or *multi-class* techniques [60]. In multi-class techniques, the normal data is assumed to fall within a number (> 1) of clusters. Accordingly, a classifier is trained to distinguish between each normal class and the rest of the classes. If a test instance is rejected by all classifiers, then it will be considered anomalous. By contrast, under the one-class assumption, all normal data instances fall within one class. Training a one-class classifier typically requires the determination of the discriminative boundary of the normal class. Any test instance lying outside the normal region will be declared to be anomalous. The majority of classification based approaches are based on Bayesian Networks, Connectionist techniques and Support Vector Machines.

Bayesian Network Based Approaches

Bayesian Networks (BNs), or *belief networks* belong to the family of probabilistic graphical models [123]. Such networks are used to represent knowledge about uncertain domains. In particular, each node in the graph represents a random variable; the edges between nodes represent probabilistic dependencies amongst the random variables. BNs have been employed to address multi-class anomaly detection problems. The naïve Bayesian approach to anomaly detection consists in estimating the posterior probability of observing a class label, given a test data instance, for each class label. The class label with largest posterior probability is chosen as the one to which the test instance belongs [38]. The prior probability for each class and the likelihood of observing the test instance given a class, are estimated from the training set. The naïve Bayesian assumption consists in considering the attributes or features of a data point conditionally independent, given a class. More complex Bayesian Networks methods have been proposed in order to account for conditional dependencies between attributes [29].

Connectionist Approaches

Connectionist strategies to anomaly detection include all neural network-based techniques used to model the normal data instances. To that end, the network is first trained to learn the boundaries between normal classes, or the distribution of the normal class. The trained model is then used to accept a test instance as normal or reject it as anomalous. The connectioninst approaches most commonly used for novelty detection are *Multilayer-Layer Perceptrons*, *Radial Basis Functions*, *Hopfield Networks*, and *Self Organizing Maps*.

Multi-Layer Perceptrons (MLPs) are neural networks with one or more layers of neurons sitting between input and outputs neurons. The goal of training the network is to find a good approximation to the actual mapping between feature vectors and the class to which they belong. As Bishop [11] pointed out, one should expect this mapping to be accurate in dense regions of the input space. Conversely, if a data point falls in a region with low density then it is likely that the point comes from a class that is not represented by the training data. In this latter case, the output of the network should be rejected as unreliable. A different MLP approach to novelty detection involves the use of *Replicator Neural Networks* (RNN) [57]. An RNN is a multi-layer perceptron where the number of input and output neurons is the same. The choice on the number of hidden layers and hidden neurons is however arbitrary. Training an RNN is equivalent to forming an implicit model of the training data. During testing, the input data instance is "reconstructed" using the trained network. The distance between actual and reconstructed input is used as a novelty score.

Radial Basis Functions (RBFs) represent another type of neural network that have been used for novelty detection. In RBFs the activation of hidden units is determined by the distance between inputs and *center* vectors [11]. The position of centers can be computed through unsupervised clustering (e.g. k-means), as has been shown by Fredrickson et al. [44]. Anomaly detection in RBFs can be carried out in many different ways. We report here the strategy suggested by Li et al. [85], consisting in assigning the test vector to the class represented by the best matching neuron. For instance, a network of two output neurons can be used to discriminate between normal and faulty events. After training, inputs lying within the region of the corresponding neuron will result in high values for that neuron and low values for the other neuron. Classification of normal versus anomalous events can then be performed via selecting the neuron with highest value. When the output of both the *normal* and *faulty* neuron exceeds a threshold, then the related input is interpreted as *unknown*.

Hopfield Networks (HNs) or Associative Networks are recurrent Neural Networks, that is to say, networks whose output is fed back to the input units using feedback connections [119]. These networks have been shown to have good novelty detection properties [67]. Two strategies can be followed when using HNs [13]. One approach consists of having a single output neuron win N inputs, where N is the dimensionality of the input space. The output neuron takes values between -1 (inactive state) and $+1$ (active state). When the Hebbian rule [59] is used to update the weights of the inputs to the neuron, the average output (membrane potential) value for stored (normal) patterns is 1 while for novel patterns is 0. Therefore, novelty detection can be performed by using a threshold of 0.5 on the output neuron. This approach works well if the noise is smaller than the absolute value of the threshold. The second approach based on HN uses the energy function to discriminate between normal and anomalous instances. As Bogacz et al. have shown [13], the value of the energy function is typically lower for stored (normal) patterns and higher for new patterns. If the noise has zero mean, a known pattern will yield an average energy value $2E = -N$ (where E is the network energy and N is the number of input neurons); whereas for a novel pattern $2E = 0$. Therefore, by using a threshold of $-N/2$, normal data instances can be effectively discriminated from anomalous observations.

Self-Organizing Maps or Kohonen networks [77] represent an alternative method for statistical clustering. They operate in an unsupervised manner and produce a low-dimensional, discretized representation of the training data. As for any neural network, SOMs are composed of neurons and weights associated to neuron inputs. Weight vectors are of the same dimension as the input data. Neurons are usually organized in hexagonal or rectangular grids. The Self Organizing Map defines a mapping between the input space and lower dimensional map space defined through the grid of neurons. Such a mapping is generated through training, by which the neuron weights are adjusted so as to represent the topological structure of the input manifold. After training, similar input samples will be mapped close together, dissimilar apart. One early approach to novelty detection via SOMs was presented by Harris [55]. This approach consists in first training a SOM on normal data instances. Once the map has organized itself, neurons will represent the classes or reference vectors of the input data. During testing, the distance between a test vector and all neurons can then be computed. The distance to the closest neuron — often referred to as *Best Matching Unit* — can be used as a measure of novelty. In SOMs, the topology of the input space is preserved as opposed to using some other unsupervised clustering algorithms, such as the k-means algorithm. This enables the definition of a more confident anomaly score. A different use of SOMs for novelty detection is the one presented by Emamiam et al. [36]. The authors showed that a novel input can be detected by observing the sequence of activated neurons in the SOM. After the map is trained, it is expected that anomalous transient inputs will activate different nodes of the SOM, with respect to normal stimuli.

Other connectionist contributions to novelty detection include *oscillatory networks*, based on models of neocortical computation [61]; *habituation approaches*, aiming at imitating the behaviour of the brain when learning to ignore repeated stimuli [95]; and *competitive learning trees* performing novelty detection on the basis of the adaptive density estimation of the input data [96].

The popularity of connectionist approaches to novelty detection is perhaps to be attributed to the very limited number of parameters that need to be optimized for training. Also, connectionist techniques do not require the distribution of the data instances to be known *a priori*. However, the best number of units (neurons) and training technique are always dependent on the problem and the data at hand. It is known that small networks have difficulties in learning, while large networks may over-generalize (overfit the training data). To address this issue, a number of techniques have been developed such as *pruning*, *regularization* and *early stopping* [94].

Support Vector Machines Based Approaches

Support Vector Machines (SVMs) are techniques used to determine hypersurfaces in any dimensional space, for separating data into clusters [140]. Hypersurfaces are selected in such a way so as to minimize the distance from them to the nearest data point on each side of the separation. A simple SVM method to one-class novelty detection has been proposed by Campbell and Bennett [19]. The authors model the data distribution through a binary function, which is positive in the normal regions of the input space and negative elsewhere. To achieve this, separating hypersurfaces are defined to be positive on one side and negative on the other. The aim of such a technique is to find a surface to wrap the entire training data. Any test instance outside this surface can be considered as novel. This basic technique is used in numerous application domains and has been extended to detect anomalies in temporal sequences [91].

3.3.2 Nearest Neighbor Based Approaches

Nearest Neighbor based anomaly detectors are based on the assumption that normal data points are more closely packed with respect to anomalous data instances. The density estimation of the neighbourhood of a point it is achieved via measuring the distance from the point to its nearest points. For non-categorical, low dimensional feature spaces, the Euclidean distance is often used [133]. However, the Euclidean distance is not the only possible similarity measure. In fact, the Euclidean metric requires the input space to be isotropic, which is rarely valid in practical applications [143]. Nearest neighbor based techniques have been proposed, based on pseudo-distance measures (i.e.

non-metric distances). These distances can be predetermined or learned in such a way as to maximize the performance of the classifier [147]. Finally, distances can be also defined in hybrid feature spaces, composed of both continuous and categorical data, as shown by Otey et al. [106].

Nearest Neighbor based anomaly detectors often use the k-nearest neighbour algorithm (kNN) to determine the anomaly score of a test instance. In particular, an anomaly score can be defined as the distance from the test data to the k^{th} nearest neighbor in the training set. With this basic approach, k is often set to 1 and a threshold is applied to the anomaly score to decide whether the test instance is anomalous or not [52]. Yang et al. used this approach for document classification [150]. The authors defined novelty between documents on the basis of some similarity measure (e.g. cosine similarity score). As a new document became available, its content was compared to all the previously stored documents (document history). This was achieved through topic-specific stop word removal, weighted use of named entities and topic-sensitive feature weighting. If the nearest neighbour to the new document, in the document history, had a similarity score below a threshold, the document was labeled as novel. Otherwise it was labeled as old. The threshold was set using cross-validation.

An alternative to using k (the index of the nearest neighbour to the test point) as a parameter for the detectors, is to select a fixed number of test points with the largest anomaly score instead, and classify them as anomalous [117]. Yet another way of performing kNN anomaly detection consists in counting the number n of nearest neighbours of a test point that are within a distance s. The number n can be seen as a density measure of the training set around the test point. The inverse of n can then be used as anomaly score for the test instance [76]. Finally, Some nearest neighbor based techniques partition the training set prior to classification, for reducing computational complexity. To that end, only the candidate partitions (i.e. the partitions that may contain outliers) are retained for further computation [117].

3.3.3 Statistical Approaches

Statistical Anomaly detectors operate under the general assumption that normal data points lie within high probability regions of the feature space, whereas anomalies occur in low probability regions. In order to determine regions of high and low probability, statistical models are first fitted to the training data. Inference is therefore made for unseen data samples, in order to determine if the instance belongs to the trained model or not. These models can be built via both parametric and non-parametric techniques, which shall be discussed in the following sections.

Statistical Parametric Approaches

Statistical parametric methods assume that the underlying distribution of the normal data is known and can be fitted to the observations via estimating the distribution parameters. The inverse of the probability density function, computed on a test instance, gives the anomaly score; that is, the less likely the observation — given the density function — the bigger the anomaly score for that observation.

Simple statistical parametric techniques consist in constructing Gaussian distributions, with single or multiple kernels, from the training data [93]. In such cases, parameter (means and variances) are commonly determined using *Maximum Likelihood Estimates* (MLE). As Roberts and Tarassenko [118] showed, the number of Gaussian kernels can be decided automatically, on the basis of the (Mahalanobis) distance between a training vector and each Gaussian of the model. The anomaly can then be the distance of the test data instance to the estimated mean (or the closest of all means). The standard deviation can be used as a threshold to discriminate between normal and anomalous data instances [129]. For example, all test instances that are more than 3σ away from the closest mean μ can be declared as anomalous. The problem with Gaussian models, especially Gaussian Mixtures Models (GMM), is that they require a very large number of samples when the dimensionality of the data is high.

Box plots have also been explored for detecting univariate and multivariate anomalies [141]. Through the 5-point summary minimum (min), lower quartile (Q_1), median, upper quartile (Q_3) and maximum (max)) provided by box plots, it is possible to distinguish normal data from anomalous instances. For example, the difference between the upper quartile and lower quartile $IQR = Q_3 - Q_1$ (referred to as *Inter Quartile Range*) can be used to determine outliers. Points lying more than $1.5\,IQR$ above $Q3$ or below $Q1$ can be considered anomalous, under the Gaussian distribution assumption for the data set.

Hidden Markov Models (HMM) are another parametric model which can be used for novelty detection [23]. HMMs are temporal probabilistic models in which the state of an event is described by a single discrete random variable. The possible values of the variable or observations are functions of the possible states of the model which are said to be *hidden* because their sequence is not directly observable. To estimate the parameters of a HMM for modeling a normal system behaviour, an *expectation-maximization* algorithm (EM) is used [123]. This algorithm estimates the model parameters upon sequences of normal events from the training set. A trained HMM can return the probability for the model to generate a given observation sequence. The probability measures obtained from the test data instances can therefore be used as anomaly score.

Statistical Non-Parametric Approaches

Statistical non-parametric methods do not make any assumptions about the model structure (e.g. Gaussian). The density function of the data set is determined from the training data, through assumptions that are typically fewer in number, with respect to parametric techniques [23]. Examples of non-parametric anomaly detectors are *Histogram* based and *Kernel Function* based techniques.

The simplest non parametric technique consists in building histograms to describe the profile of the normal data. Histograms *bins* denote the frequency with which features or attributes of the data are observed. For univariate data, the histogram based technique checks if a test instance falls in any one of the histogram bins. Depending on the bin height (frequency), an anomaly score is returned. Histogram based approaches can also be applied in the case of multivariate data. In this case, a per-attribute anomaly score is computed as the height of the related histogram bin. The overall score results from an aggregation of the various per-attribute scores [23]. In general, as noted by Chandola and Kumar [23], the amount of bins used is a key factor for anomaly detection. Indeed, too many bins could produce many false alarms; too few bins may result in too many misses.

The Parzen windows method [35] represents another non-parametric data density estimation approach. In particular, the actual density function is approximated via Kernel functions[1]. This approach is very similar to that of fitting Gaussian distributions to the normal data. The difference in this method is the density estimation technique. In statistics, kernel density estimation is considered a non-parametric method for estimating the probability density function of a random variable. Similar to the parametric approaches, a test instance is classified as anomalous if it lies in the low probability region of the computed density function.

3.3.4 Information Theoretic Anomaly Detection Techniques

The last family of anomaly detectors concern *Information Theoretic* techniques. Information theoretic approaches are based on the assumption that anomalies increment the complexity of the dataset. Common complexity measures used are the *Kolmogorov*

[1]A Kernel is a symmetric but not necessarily positive function that integrates to one.

Complexity, the *entropy* and the *relative entropy*. Formally, given a dataset D of complexity $\mathcal{C}(D)$, the problem of finding anomalies in D is posed as finding the minimal subset, I, of D such that $\mathcal{C}(D) - \mathcal{C}(D - I)$ is maximum [23]. The instances in I thus obtained are regarded as anomalous. This problem of finding I is, in effect, a dual constraint satisfaction problem. In particular, the size of I needs to be minimum and the complexity of the reduced data set is maximized. Optimal solutions to this problem require an exhaustive search among all possible subsets of the data set; this makes the basic technique unappealing. However, heuristics have been proposed in order to reduce the search time [3].

3.4 Object Recognition in Real and Synthetic Images

As it will be shown in later chapters, in some cases, the detection of unintended artifacts in synthetic images is best posed as an object recognition problem. This is the case, for instance, of anomalies featuring highly characteristic spatial or spatio-temporal patterns in image space. In image processing and computer vision research, the detection of objects in real images it typically considered to be a hard task. The difficulty is due to many factors such as the continuous change of 2D object shape depending upon the angles under which the object is observed; the change of light conditions and therefore of colors over the object surface; and the object clutter and occlusions from the rest of the environment. For these reasons, it becomes hard to find robust representations suitable to model generic objects. Yet, the problem of object recognition is critical in many important application scenarios. Over the fairly recent history of machine vision, a large body of algorithms has been developed to address the object recognition problems. Such algorithms are typically divided into three major categories; these are *model based*, *shaped based* and *appearance based* approaches, which shall be reviewed shortly. Almost all techniques base their behaviour on the common principle of describing images and objects by sets of feature vectors. These vectors can be composed simply by raw pixel intensities or they can represent other more complex properties of the image. However, it is generally agreed that good descriptors should exhibit some invariance property (e.g. invariance to illumination change, noise and geometric transformation). Ultimately, the task of object detection is to assign a set of feature vectors to one of the classes or labels denoting the various objects to recognize. Often, this is accomplished by finding a correspondence between the image descriptors and comparable features of the model used [114].

3.4.1 Model Based Approaches

In model based schemes, the knowledge of the appearance of an object is provided by an explicit 3D model of its shape, defined either as a collection of three dimensional primitives [75], or through point clouds [97]. The task of the majority of model based approaches is to correctly identify objects in a scene and estimate their pose, that is, location and orientation, even in the presence of clutter and occlusion. The 3D object recognition paradigm is often an appealing alternative to 2D recognition techniques, for objects undergoing illumination, scale and pose change as well as occlusion and shading. Three-dimensional models of the objects are typically constructed offline, from multiple 2D visualizations, and stored in a database. During recognition, test images are converted into the same representation and matched with the database models [97]. The challenge of model based approaches is the reconstruction of the 3D geometry from the set of (often unordered) overlapping views. To that end, correspondences need first to be found among the images. Correspondence techniques range from pair-wise approaches [25], to the less computationally expensive multiview algorithms [97]. Through a correspondence process, the 3D object is registered in a common coordinate basis. Recognition is then performed through feature matching, by using *B-Splines* [26], *High Order Tangents* (HOT) curves [70] and *Spin Images* [68], amongst others. The assumption the model based scheme makes is that a three dimensional model (e.g. a set of connected vertices or a point cloud) of the target object is available or can be extracted during both training and detection, from example, from 3D scanner data.

3.4.2 Shape Based Approaches

In shape based approaches, objects are recognized by either their shape [105; 138] or contour [48]. As with the model based scheme, input objects are assumed to be in some 3D representation, at least during the training phase, to form the database or the objects. Shape descriptors can then be extracted either in 3D space (i.e. through spin images [68]) or in image space (i.e. through SIFT points [89]). Recognition is performed through descriptor matching or silhouette matching, depending on the way the object is described. When the number of descriptors is large, *codebooks* are typically generated by partitioning the feature space. In such cases, the object will be described by histograms representing the frequency with which features appear in each visualization. For highly complex scenes (e.g. urban environments) it has been shown that the detector accuracy can improve if contextual information (e.g. the object position with respect to its environment) is included into the object descriptors [48].

3.4.3 Appearance Based Approaches

In appearance based methods, the objects are modeled through feature vectors (descriptors) extracted from a set of images. Recognition is performed by matching the feature vectors of the new input image to the model set [121]. This can be accomplished through either *local* or *global* approaches, depending on the type of features used to describe the objects of interest. Local features, describe properties of small, local image patches. Such descriptors are selected to be invariant to illumination change, noise and affine transformation changes. To endow a descriptor with all these properties, typically, various simpler descriptors are combined into a single more complex feature vector. Conversely, global features relate to the entire object or image. Examples of global descriptors are histograms of features (such as colors and gradients), and vectors resulting from dimensionality reduction methods, such as Principal Component Analysis (PCA) [69], Independent Component analysis (ICA) [63] and Non-negative Matrix Factorization (NMF)[83]. Dimensionality reduction of high dimensional images onto subspaces is very common in object recognition and pattern recognition tasks in general. Working with lower dimensional feature spaces has the enormous advantage of reducing issues like data *overfitting*, *curse of dimensionality* and *computational complexity* [130]. In order to classify an object through global features, template matching techniques are typically used [121]. The word *template* is used to indicate the representation — typically through vectors — of the target object or scene. Given a set of templates representing an object, any unknown sample can be classified by finding the correlation between the new template and the set of *training* templates. If the correlation is above some threshold for at least one training template, then a match is found. As Roth and Winter pointed out [121], global approaches are considered to be more robust than local approaches. However, local methods represents the entire object locally, hence, they are more effective at coping with partial occlusions and clutter.

As far as local approaches are concerned, local features are computed over *interesting points* of the images, that is to say, points in the image that can be found in some highly repetitive manner. The most popular algorithms for computing interesting points are Harris-based, Difference-of-Gaussian (DoG) and Hessian affine invariant point detector, amongst others [121]. The idea behind such detectors is to find, in the image, regions where the (first or second) derivatives along both x and y directions have high values. The rotational invariance property of such descriptors is achieved by considering only the values of the two principal image gradients (curvatures), rather than their direction. To make the description scale invariant, gradients are computed at different integration scales. The local extremum of the descriptor's response with respect to the integration scale is used as scale selection criterion.

Once the interesting points of the image are computed, invariant feature descriptors are extracted from small image patches centered at the interesting points. The most common feature descriptors are *Scale-Invariant Feature Transform* (SIFT) [89], *Principal Component Analysis*-SIFT (PCA-SIFT) [73], *Gradient Location and Orientation Histogram* (GLOH) [100] and *Speeded Up Robust Features* (SURF) [9]. Object recognition is then performed by matching the features from the test image with the features in the database, collected during training. Matching is typically performed through three different strategies; these are *threshold-based* matching, *nearest neighbor-based* matching and *distance ratio-based* matching [100]. In the case of threshold-based matching, two interesting points from two images (e.g. the test image and the database image) are matched if the Euclidean distance between the related descriptors is below a threshold. With the threshold method a descriptor can have several matches. In order to make the matching more accurate, typically the nearest neighbor-based method is used, in which two points A and B are matched if the descriptor D_B is the closest to D_A in the Euclidean sense, and the distance between them is below a threshold. The third type of matching is similar to nearest neighbor matching, except that the threshold is applied to the distance ratio between the first and the second nearest neighbor. Thus, two points A and B are matched if $\|D_A - D_B\|/\|D_A - D_C\| < t$, where D_B and D_C are the first and the second nearest neighbour to D_A, and t is a threshold.

3.5 Summary

In this chapter we have reviewed the underlying theory involved in the pattern classification approaches of novelty detection and object recognition. Some of the techniques presented here will be incorporated into the consistency detection solutions presented in subsequent chapters. In particular, the object representations commonly used in model based object classifiers will be similar to the one we will use in Chapter 4. Novelty-detection techniques based on Artificial Neural Networks and Self-Organizing Maps will be used to build robust inconsistency detectors in Chapter 5. Finally, the appearance based classifier we will present in Chapter 6, will be based on a common point descriptor (the *Harris* descriptor) used in appearance based object recognition.

4

Model Based Classifiers

In the previous chapter, we explained the difficulties of current object recognition approaches in accounting for severe geometry and color changes in real objects. Fortunately, when dealing with rendering systems we can exploit extra information available from the virtual world as the scene is rendered. The anomaly detection approaches presented in this chapter are all based on a fundamental concept common to the vast majority of 3D rendering systems. While the visualization of an object may change in screen space (as it does in real images) its geometric description never changes in local space. This property will be clarified in the next section.

The aim of this chapter is to show that, under circumstances in which the light contribution from the environment is mild and the state of the lights sources does not greatly vary, it is possible to make plausible inferences about the appearance of the objects by exploiting their canonical model based representation, while ignoring the color synthesis process used by the rendering system.

4.1 Stages of Image Synthesis

The vast majority of current rendering systems synthesize images through stages. These stages operate in a pipeline, that is, in parallel and in a fixed order [39]. Each stage receives its input from the prior stage and sends its output to the subsequent stage (Figure 4.1 (a)). This mechanism assumes virtual entities to be represented through meshes. The vertices of a mesh are assumed to be in *local* or *object space*, the local coordinate system in which the 3D object is defined.

Virtual objects, in their original *local* space description, are passed from the application to the graphics pipeline. The screen position of the input vertices is then computed

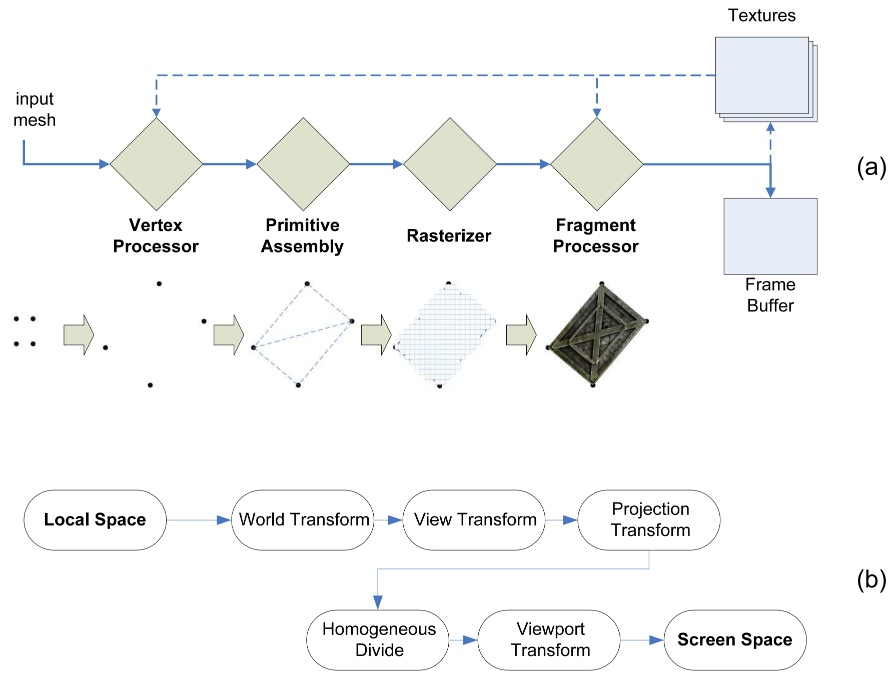


Figure 4.1: Stages of image synthesis - The final 2D appearance of an object is the result of a number of stages in the graphics pipeline. These are the per-vertex operations, which transform the vertices from local to clip space according to the instructions written by the programmer. The primitive assembly then generates triangular faces from the list of vertices defined in screen space. Next, the faces are converted into pixels by the rasterizer. Finally, pixels are colored according to the fragment shader (code) written by the 3D application programmer (a). The local to screen space transformation is carried out partially by the vertex processors (world, view and projection transformations), and partially by the fixed pipeline (homogeneous divide and viewport transformation), before the primitives are passed to the rasterizer (b).

according to the world, view and projection transformation and other transformation parameters passed by the application. Specifically, the *World* and *View* stages of the pipeline are the affine geometric transformations that give the position in *eye* or *camera* space. To perform affine transformations through matrices, a homogeneous component, w, is introduced so that any vector (x, y, z) becomes (x, y, z, w), where w is typically set to 1. The *Projection* transformation then gives a position in a coordinate system bounded by the homogeneous unit cube: the *clip* space. The clip space position of a vertex of original position $(x, y, z, 1)$ is therefore be computed as follows:

$$[x', y', z', w] = [x, y, z, 1]WVP = [x, y, z, 1]M_{wvp} \tag{4.1}$$

Here, the 4×4 matrix M_{wvp} results from the multiplication of the world, W, view, V, and projection, P, matrices. Such per-vertex operations are typically carried out by the *vertex shaders* (processor), which run user-defined graphics code for manipulating the original object geometry. Next, the position in Normalized Device Coordinates (NDC) is given by the *Homogeneous Divide*. This step brings the homogeneous component back to 1 after carrying out the projection matrix multiplication. Finally, the *Viewport* transformation stretches and translates the projected coordinates to fit a specific position on the screen (Figure 4.1 (b)). Once the vertices have been computed by the vertex processors, the primitives are assembled and the boundaries *filled in* with pixels by the *rasterizer*. It follows, that the final shape of any virtual object is ultimately decided by the World, View and Projection matrices, which map from an invariant canonical local space representation to the final shape of the objects on screen.

The color distribution over the surface of an object, from a physics viewpoint, is a property of the object material and geometry, of the light hitting the surface and the angle under which the object is observed. As discussed in Chapter 2, rendering systems perform real-time color synthesis, via texturing and local and/or global illumination techniques which are typically per-pixel operations implemented via *fragment shaders* [90]. The image thus generated can be either written to the frame buffer memory and displayed or it can be written to texture memory (Figure 4.1 (a)). The *render-to-texture* mechanism is the way current GPUs implement direct feedback from output to input without passing the data through the CPU again [54]. Having access to the output from the pipeline is useful for performing random-access to fragment and vertices as localized *gathering* operations. Motion blur [120] and Shadow Mapping [82] are two examples of visual effects that base their functionality on the render-to-texture mechanism.

GPUs also feature mechanisms designed to avoid computing the color of pixels that will not be seen. One of these is known as *Z-culling* [112]. This technique consists in comparing the depth (z value) of the input fragment with the depth of the fragment already present in the *frame buffer*. If the incoming fragment fails the depth-test (or z-test), it is discarded before the pixel color is computed in the fragment processor. The depth value of all pixels in the frame buffer is stored in an array called a *depth buffer* or *Z-buffer*. The z-test is part of a set of steps known as *raster operations*.

4.2 Defining Visual Consistency

The aim that we wish to fulfill is to exploit the canonical space representation of virtual objects for assessing both geometry and color consistency. In particular, we want to re-construct the original geometry and color of the objects, given their correct screen space appearance perceived by the user, throughout a play session. To this end, the screen space information will need to be extracted from *trusted* or validated frames, that is, images that have been previously assessed against visual inconsistencies — for example, by the designer or the developer of the environment. To re-construct the original object appearance, we will need to invert the geometric transformations carried out by the rendering system (Figure 4.1 (b)), thus bringing both screen shape and color back to local space.

We call the models of the object created in such a manner *colored point clouds*. The reason being that, as we shall see shortly, our objects will be represented by 3D colored points, forming cloud-like structures in object space. Since the validated frames are assumed to be correct, the clouds will constitute the ground truth database for our system and will be used to make inferences about new visualizations. Figure 4.2 depicts the general process of visual inconsistency detection we aim to build. As the information we use to make inference concerns the (3D) models of the objects, we call this approach *model based classification*.

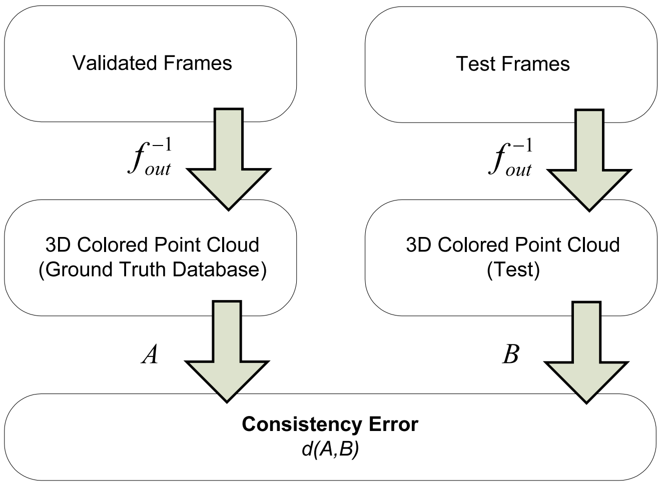


Figure 4.2: Model based classifier scheme - The diagram shows the process of measuring consistency through 3D point cloud structures. Test images are first converted to point clouds in object or local space, encoding the original spatial and color information of pixels. This is done by using the same rendering system f_{out} used by the virtual environment. As new objects arrive, their point cloud is computed and compared with the ground truth database in order to establish correctness.

4.2.1 Colored Point Clouds Synthesis

As stated earlier, the object space representation of our target objects can be generated by inverting the geometric transformations carried out by the rendering system. Assuming we have observed n pixels of the object, for a given frame, let P_s be the $n \times 4$ matrix, with row i equal to $(x_i, y_i, z_i, 1)$, containing the screen space coordinates of all such pixels; and M_{wvp} the 4×4 world-view-projection matrix transforming the object geometry from object space to screen space. Moreover, let M_r be the 4×4 viewport matrix of the observed frame; T the $n \times 4$ matrix of non-scaled object space coordinates relative to P_s; and "./" the operator denoting an element-wise division. The following relations hold:

$$\begin{aligned} T &= P_s \times M_r^{-1} \times M_{wvp}^{-1} && (4.2) \\ P_o &= T./W && (4.3) \end{aligned}$$

where P_o is the scaled object space version of P_s, that is, the matrix containing the object space coordinates of the points equivalent to the pixels of coordinates P_s. P_o is of the same dimensions as P_s. The matrix W contains the homogeneous coordinate components of T; the columns of W are identical to each other and equal to the last column of T.

Since we want to measure both geometry and color consistency, we will also need to include the pixel colors into our description of virtual objects. To that end, we append a new matrix C next to P_o; such a new matrix will denote the final color of the pixels on screen. This process produces the combined $n \times 7$ matrix $\mathcal{D}$, whose row i is equal to $(\mathbf{p}_i, \mathbf{c}_i)$. Here, $\mathbf{p} \in P_o$ denotes the position of the pixels in object space and $\mathbf{c} \in C$ is a vector of dimension 3 whose column components indicate the color of the pixel (e.g. in RGB space) as it appears on screen. The matrix $\mathcal{D}$ is what we shall refer to as the colored point cloud.

Throughout this chapter we shall use the terms *points* and *pixels* interchangeably. Note, however, that a rectangular pixel of a certain position (x, y) and depth (distance camera-pixel) z, in screen space, projects to a three-dimensional volume, in clip space, as shown in Figure 4.3. Similarly, all clip space points of an object lying within such a volume will always project to the same pixel. The size of such a volume is decided by the resolution of the buffers used to store the screen positions of pixels and their depth. The coarser the resolution of the buffer, the more information will be lost about the original geometry of the objects, the less accurate will be our point clouds in representing the actual object geometry. Later in this chapter, we will see the implication of this discretization problem.

The reader may have noticed that the perspective-projection transformation — encoded in the world-view-projection matrix M_{wvp} — is singular due to the loss of the z dimension; individual points in eye space that lie along the same line of projection will project to a single point. This is true, as long as the 3D object space coordinates are transformed into 2D pixel coordinates. However, the projection carried out by rendering systems preserves the z component. Such a value becomes the depth of the pixel and is stored in a data structure often referred to as the *depth* or Z buffer (see Figure 4.3). Finally, note that the rescaling operation performed by Equation 4.3 should take place after inverting the Projection transform and before multiplying by the inverse of the World-View matrix. However, because no affine transformation (World and View) can alter the homogeneous component of a coordinate vector, this operation can be performed at the end of the reversing process. The advantage of postponing the rescaling operation is to compute the inverse of two matrices (the Viewport and the combined World-View-Projection) instead of three (Viewport, Projection and the combined World-View).

4.2.2 Visual Consistency Definition

By applying Equation 4.2 and 4.3 over a set of frames we effectively build a canonical 3D representation of color and geometry for the target object (Figure 4.4). The information contained in this representation depends on the conditions (camera angles and lights) under which the object is rendered across the frames. Once the point clouds have been built, consistency can be assessed through them. To that end, we propose the following general definition of consistency between colored point clouds:

Definition 4.1. *Given a colored point cloud $\mathcal{D}'$ of an object relative to a single frame and a colored point cloud $\mathcal{D}$ of the same object relative to some collection of frames; we say that $\mathcal{D}'$ is ε consistent with $\mathcal{D}$ if*

$$d(\mathcal{D}', \mathcal{D}) \leq \varepsilon \tag{4.4}$$

We call $d(\mathcal{D}', \mathcal{D})$, the *visual consistency error* of $\mathcal{D}'$ relative to $\mathcal{D}$. Here d is some distance function and ε is some (small) constant. If $\mathcal{D}$ is the ground truth database for the target object and $\mathcal{D}'$ its colored point cloud from the new frame, we say that the object of the new frame is visually inconsistent if $d(\mathcal{D}', \mathcal{D}) > \varepsilon$. In the coming sections,

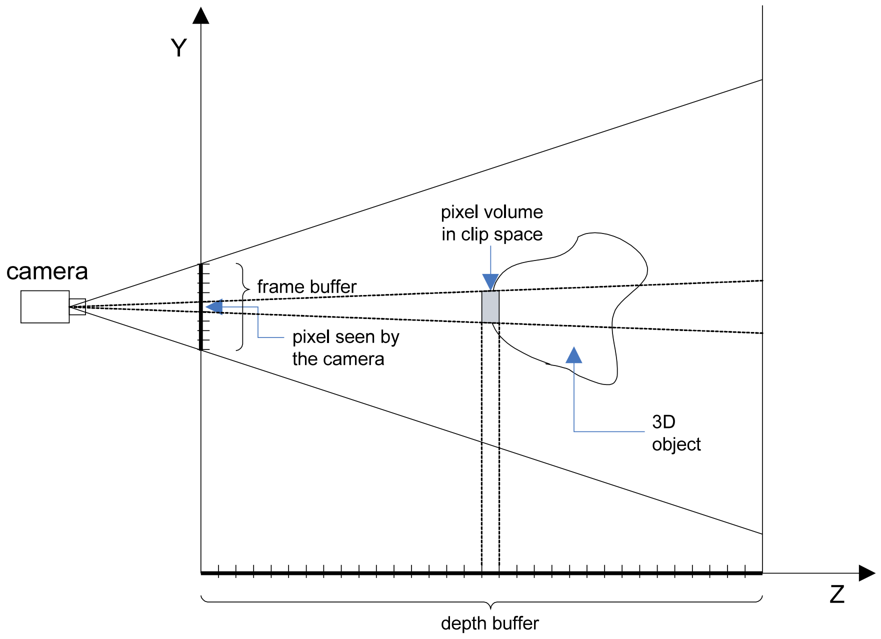


Figure 4.3: Pixel projection in clip space - During the geometric transformation operations of the graphics pipeline, the position of points in object space is converted into screen space coordinates (x, y, z); the (x, y) components indicate the position on screen, the z coordinate denotes the distance of the 3D point from the camera (in the figure only the YZ plane is considered). Because such coordinates are computed and stored into buffers of finite resolution, discretization errors are introduced. In particular, all points of an object within the same pixel volume will project to the same pixel coordinates in screen space.

we will show how to build the ground truth database $\mathcal{D}$ and how to measure the error d in both object space and screen space.

4.2.3 Some Necessary Epistemological Considerations

The inconsistency detectors we will present in this and the next chapter, are based on the assumption that a number of *validated* views of the target objects are available, that is, views that are perceived as consistent by some observer. This assumption enables us to treat consistency merely as a physical property of the environment; the subjective component of what is right or wrong in the scene is implicit to the choice of the observer when selecting the consistent frames.

Furthermore, the consistency will be assessed at the level of individual entities, rather than the scene in its wholeness. This will exonerate us from performing the analysis at different scales. Indeed, it is important to point out that even though, at a low level, the information from all parts of an object may appear consistent with the

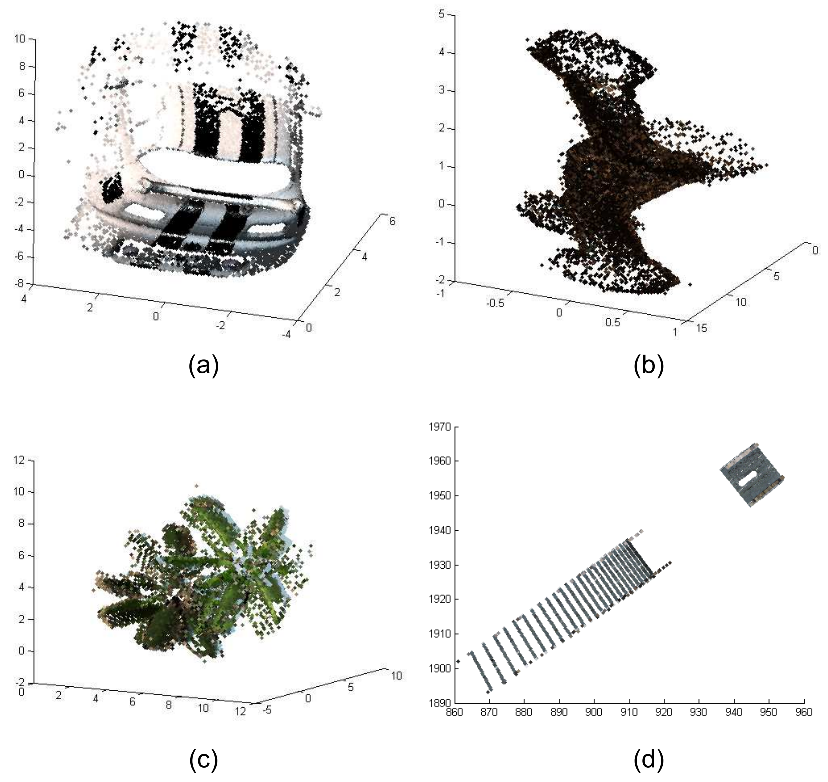


Figure 4.4: Examples of Colored Point Clouds - 3D representation of the colored point cloud $\mathcal{D}$ for four different objects observed in a collection of frames; a car (a), the trunk of a palm tree (b), the leaves of a palm tree (c) and a segment of the track (d).

user expectation, at a higher level, the parts may appear in an odd mutual relation. As an example, imagine a car whose wheels, windows and body are properly visualized but in the wrong position or size. Finally, note that even if an object in its wholeness may appear as consistent, its relation with the rest of the scene may not.

4.3 Building the Model through the Game Engine

From the definition of consistency given earlier, it follows that in order to decide whether an object is visually correct or not we first need to compute the matrix $\mathcal{D}$ from some given collection of frames in which the object appears as correct. Such a process requires P_s and M_{wvp} to be known for each one of the validated frames. The viewport matrix M_r is, in general, constant throughout the play session.

During the rendering process, a graphics application passes the vertices that need to be rendered to the GPU, as well as their geometric transformation matrices, amongst other things. The screen space coordinates of the pixels are then computed by the GPU. Therefore, to determine both P_s and M_{wvp} for each object, we need the GPU to disclose which pixels have been rendered by which matrix. This can be done by making the GPU produce one additional image in which colors denote the geometric transformation that the pixels have just undergone. We call such an image the *matrix map*. One way to generate the matrix map through *shader code*[1] is described in Algorithm 4.1.

Algorithm 4.1: Shader code used to generate matrix maps

```
struct VSOutput
{
  float4 Pos : POSITION;
  float2 Depth : TEXCOORD0;
}
VSOutput OMMap_VS(float3 Pos : POSITION)
{
  VSOutput outVS = (VSOutput)0;
  outVS.Pos = mul(float4(Pos.xyz, 1.0f),worldViewProjMat);
  outVS.Depth = float2(outVS.Pos.z,outVS.Pos.w);
  return outVS;
}
float4 OMMap_PS(VSOutput In) : COLOR
{
  float4 outPS = (float4)0;
  outPS.xyz = matColor.xyz;
  outPS.w = (In.Depth.x)/(In.Depth.y);
  return outPS;
}
```

To make the process of generating the matrix maps as game software independent as possible, we created a class, *MatrixMapShader*, that inherits from the same class used to initialize any other shader effect in the game. The purpose of *MatrixMapShader* is twofold. Firstly, it creates an interface between the game and the shader code and ensures that the objects in both the final image and the matrix map are rendered by using the same graphics data (i.e. matrices and vertices). Secondly, it produces a data structure (a hash table) which stores the matrices M_{wvp} of the objects and their identifying colors (labels). As for the other render classes of the game, our class is instantiated once and used whenever an object needs to be rendered. The class functionality is turned on and off by debug flags in the game source code. The pseudocode of such a class is shown in Algorithm 4.2.

The matrix map, denoted by the attribute matMap, is drawn through the function

[1]Recall that a shader is a set of software instructions that is executed by the GPU.

GenerateMap via the *draw* function provided by the graphics interface. When an object needs to be rendered, the game calls the function UpdateMap first, which adds the object name and matrix, along with the associated colors, to an internal hash table called matColTab. The function UpdateMap then instructs the GPU to render to the texture matMap, rather than the frame buffer, and passes the current transformation matrix M_{wvp} and its discriminative color to the shader. The function GenerateMaps, which is called by the game right after calling the function UpdateMaps, sets the shader code to use, and sends a *draw* request to the GPU to draw the current object obj. Here, the shader code is the one reported in Algorithm 4.1. Once the rendering of the object is complete, the depth buffer is *cleaned* to avoid introducing artifacts to the final image[1].

Algorithm 4.2: MatrixMapShader class

```
Attributes:
  matMap: texture, of the same size as the frame buffer
  matColor: $(R, G, B)$ array used for coloring matMap
  matColTab: hash table containing matColor arrays
  key: char array, denoting the hash table key
Methods:
  MatrixMapShader()
    matMap $\leftarrow$ new texture of the same size as the frame buffer
  UpdateMap(obj,$M_{wvp}$)
    matColor $\leftarrow$ Colorize($M_{wvp}$)
    key $\leftarrow$ Append($M_{wvp}$, obj)
    if key not in matColTab then
        add tuple $<$ key, matColor $>$ to matColTab
    end
    set render target to matMap
    set shader WorldViewProjection matrix to $M_{wvp}$
    set shader matColor to matColor
    commit changes to the GPU
  GenerateMap(obj)
    draw obj
    clean depth buffer
```

The unique color for the matrix M_{wvp} is computed by the function Colorize. In our implementation, Colorize returns the value of an internal counter incremented whenever the input matrix is novel, that is, it is not contained in a database maintained

[1]The depth buffer is a resource frequently utilized in current real-time rendering techniques. Because there is no guarantee that such a buffer is cleared before being used by the application to test, it is good practice to clean it before exiting the testing functions.

internally. The hash table matColTab is queried and updated upon a key which combines the name of the mesh to render and the string version of the vectorized matrix M_{wvp}. The operation of string concatenation and vectorization of M_{wvp} is carried out by the function APPEND. Finally, the hash table is updated only if the current key is not in the table already.

Once rendered, the matrix map will enable later algorithms to pinpoint which pixels in the image have been rendered by which matrix. More precisely, a pixel at screen position (x, y) will have an associated color in the matrix map at the same position. The associated color, thanks to the table matColTab, can be used to identify both the matrix M_{wvp} and the mesh name to which the pixel belongs.

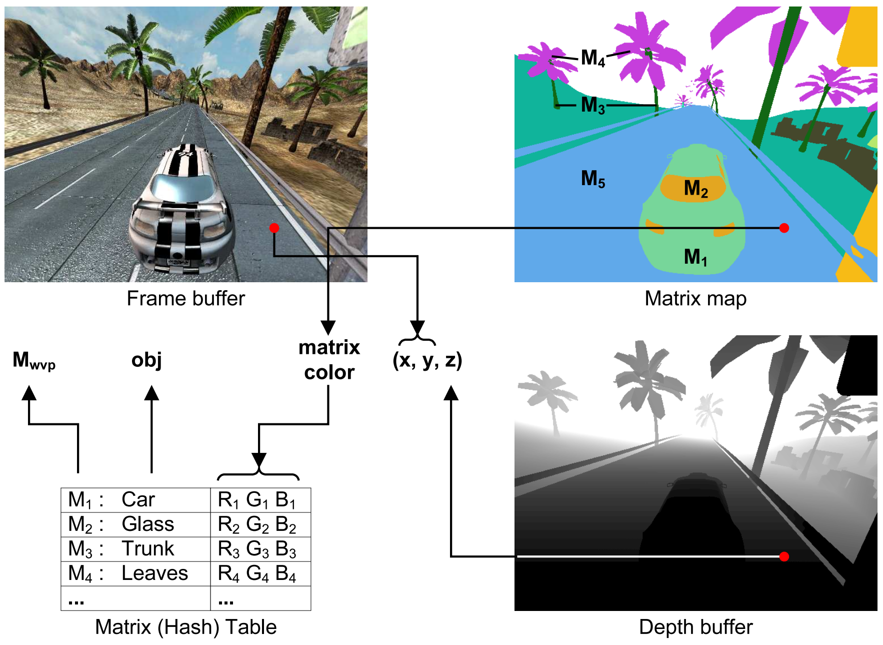


Figure 4.5: Extraction of matrix and pixel information - In order to retrieve and store M_{wvp} and P_s, a matrix map is generated for each frame. The pixel coordinates $(x, y, z) \in P_s$ are taken from the frame and depth buffer. The matrix M_{wvp} retrieved from the game engine and stored in the matrix table, along with its color. The color of the matrix is read from the matrix map.

We have seen how to determine M_{wvp} for each pixel of a frame. However, in order to compute the colored point cloud $\mathcal{D}$, we still need to determine P_s. Recall that P_s is the matrix containing the screen space coordinates (x, y, z) of all pixels of the target object. The components (x, y) are simply the position of the pixel on screen, z is the distance camera-pixel which can be read from the *depth* or Z-buffer (see Section 4.1, if available, or can be computed through the same shader used to generate the matrix map. In our work, the depth buffer is computed through the code presented in Algorithm 4.1. Specifically, the depth value is computed in the vertex shader `OMMap_VS` and normalized in the pixel shader `OMMap_PS`. The result is written to the α channel of the matrix map, the shader variable `outPS.w`. Figure 4.5 depicts an example of matrix map and depth buffer of a frame. Note that, with this implementation, the object map can map up to 2^{3d} matrices per frame where d is the color depth used. For a 32-bit *truecolor* image (8 bits per channel) the maximum number of matrices that can be stored for a single frame is $\approx 1.6 \times 10^7$.

4.4 Visual Consistency Error in Object Space

Given the consistent point cloud $\mathcal{D}$ (the ground truth database) and a new point cloud $\mathcal{D}'$ from a new frame, we are now interested in measuring the error d introduced earlier. In particular, we wish to formulate d in such a way as to make it return plausible outputs; if the target object in a new frame is visually different from the one observed in the validated frames, we should expect the error d to be big. Conversely, if the object looks similar to the one from the validated frames, d is expected to be small.

4.4.1 Geometry Errors

One of the two components of the point cloud is the matrix P_o of pixel coordinates in object space. Such a matrix is defined in the (three dimensional) Euclidean metric space. A possible choice for our geometry error is then the *Hausdorff* distance [16], defined as:

$$d_H(X,Y) = \max\{\sup_{x\in X}\inf_{y\in Y} d(x,y), \sup_{y\in Y}\inf_{x\in X} d(x,y)\} \tag{4.5}$$

where X and Y are two non-empty subsets of a metric space and sup and inf represent the *supremum* and *infimum* element of the related subset. The distance d in Equation 4.5 is, in our case, the Euclidean distance. Intuitively, if the target object from a new frame has an anomalous geometry, its anomalous points will be displaced far away from the closest points of the ground truth database $\mathcal{D}$, no matter how topologically complex the original geometry is (Figure 4.6). Thus, we can say that the geometry of a new point cloud is ε consistent with the validated point cloud, in the Hausdorff metric, if

$$d_G(\mathcal{D}', \mathcal{D}) = d_H(P', P) \leq \varepsilon \tag{4.6}$$

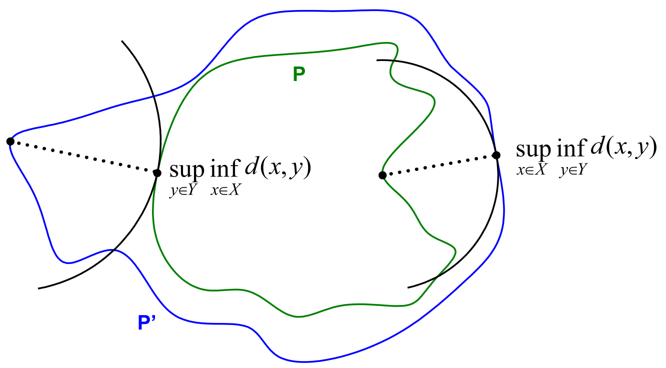

P
P'
$\sup_{y \in Y} \inf_{x \in X} d(x,y)$
$\sup_{x \in X} \inf_{y \in Y} d(x,y)$

Figure 4.6: Hausdorff distance - Components of the calculation of the Hausdorff distance between the database points P of an object and the set of points P' of the same object from a new frame.

where d_G is the geometry error of $\mathcal{D}'$ given $\mathcal{D}$; P' and P are the pixel position matrices of $\mathcal{D}'$ and $\mathcal{D}$ respectively. In order to compute the Hausdorff metric, the distance between all points of the validated point clouds and all points of the new point cloud needs to be evaluated. The validated point cloud can be extracted by an arbitrary number of frames, thus, it may contain an arbitrary number of points. The more points in the point cloud, the more memory is needed to store the database, the more computing power is required to evaluate d_G. Strategies need to be developed in order to mitigate both memory and computing complexity.

The solution we propose consists in replacing the entire point cloud with representative points or *centers*. We compute such centers via adaptively clustering the point cloud as new validated frames arrive. The representative points of the cloud will coincide with the cluster centers. This process is described in Algorithm 4.3.

Ada-Clustering accepts a new point cloud, P'_o, as an input along with the position matrix, P_o, of the colored point cloud database. Both P'_o and P_o are defined in object space. The function also expects the user to specify a threshold, ρ, indicating the maximum radius of clusters; for a point to belong to the closest cluster, its distance from the cluster center must be less than ρ. This function should be called whenever new valid pixels of the object of interest are available in order to update the ground truth database with new centers. When called for the first time, the database is obviously empty, hence a first partitioning needs to be performed with a minimum number of clusters (in our experiments, min_centers is set to 1). This operation is carried out by the function Cluster which can be any partitional clustering algorithm (in our implementation we used K-means). Next, the rows of the matrix P'_o are clustered according to the centers in P_o. This is accomplished by computing the minimum

Algorithm 4.3: Adaptive Point Cloud Clustering

```
1  function ADA-CLUSTERING($P'_o$,$P_o$,$\rho$) returns a matrix of cluster center coordinates
     inputs:
           matrix $P'_o$ of $l$-dimensional point coordinates, of size $k \times l$
           matrix $P_o$ of the colored point cloud
           cluster radius $\rho$
     locals :
           number min_centers of initial cluster centers for empty databases
           index array pixelClstIdx of clustered points, of size $k$
           array pixelClstDist of distances from points to nearest centers, of size $k$
           index array currPixelClstIdx of points belonging to the current cluster
           matrix currPointCloud of point coordinates belonging to the current cluster
           max distance cloudMaxDist from the point cloud to the current cluster center
           matrices tmpCenters and updCenters of cluster centers, initially empty
           flag updateCBook indicating whether $P_o$ needs to be updated, initially false
2    if $P_o$ is empty then
3        $P_o \leftarrow$ CLUSTER($P'_o$,min_centers)
4    end
5    for $i \leftarrow 1$ to $k$ do
6        pixelClstIdx[$i$] $\leftarrow$ index of the min value of $\|$ROW$(P'_o, i) - P_o\|$
7        pixelClstDist[$i$] $\leftarrow$ min value of $\|$ROW$(P'_o, i) - P_o\|$
8    end
9    for $c \leftarrow 1$ to number of centers in $P_o$ do
10       currPixelClstIdx $\leftarrow$ indices $j$ of pixelClstIdx such that pixelClstIdx[$j$] $= c$
11       currPointCloud $\leftarrow$ ROW($P'_o$, currPixelClstIdx)
12       cloudMaxDist $\leftarrow$ max value of the array pixelClstDist (currPixelClstIdx)
13       if cloudMaxDist $\leq \rho$ then
14           tmpCenters $\leftarrow$ ROW($P_o, c$)
15       else
16           updateCBook $\leftarrow$ true
17           tmpCenters $\leftarrow$ SPLIT-CLUSTER(currPointCloud,$\rho$)
18       end
19       concatenate new cluster centers tmpCenters to updCenters
20   end
21   if updateCBook then $P_o \leftarrow$ updCenters
22   return $P_o$
```

Euclidean distances between the elements (rows) of P'_o and the database centers (lines 6 and 7); the indices of such minimum distances will indicate the centers to which the points of P'_o belong. The algorithm is adaptive in that the number of centers of P_o may grow if the point cloud grows. The reason why the point cloud may grow as new frames are observed is related to the fact that not all parts of an object can, in general, be viewed from a single camera angle. If the new camera angle reveals new object parts, then the point cloud database will be extended accordingly. We grow the number of centers by looking for *outer* points, that is, points lying outside the sphere described by ρ. If a cluster contains outer points then its center is replaced with a number of new centers (lines 16 and 17). To this end, the iterative function Split-Cluster is called to ensure that none of the newly created sub-clusters contains outer points. The function Split-Cluster uses the same partitioning algorithm as Cluster. Once Ada-Clustering terminates, the database is updated and the point cloud P'_o from the current frame can be discarded. Because we only retain the coordinates of the cluster centers, rather than the entire point clouds, we are able to generate databases for virtually any geometry from any number of frames; the final position and number of clusters in P_o will only depend on the original size of the object and on the cluster radius ρ but not on the total number of pixels processed.

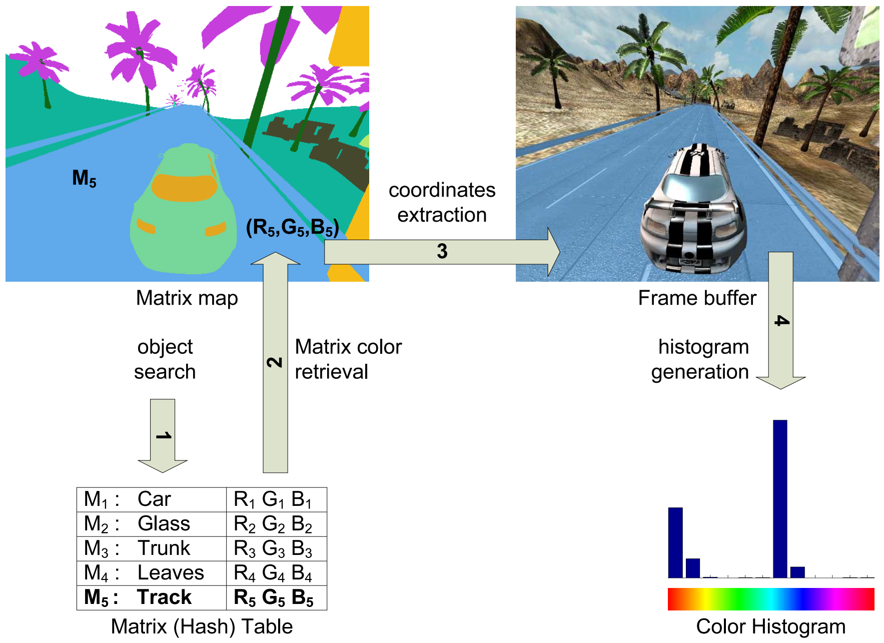
4.4.2 Color Errors

As colors too are defined in a metric space (e.g. RGB), the color consistency could as well be computed by using the Hausdorff metric through mechanisms similar to the ones we presented in the previous section. However, in our work we decided to describe color in a way that is quite common in image processing research, viz, through histograms. In natural images, color histograms are often used to represent color distributions of the entire image or small regions of it [128]. In our case, we use histograms to describe the color of the target object only.

To determine the pixel coordinates of the target object we use the matrix map and the matrix table introduced earlier, as depicted in Figure 4.7. The object color is therefore extracted directly from the frame, at the position specified by the extracted coordinates. The color consistency error can then be defined as the distance between color histograms. In our implementation, such an error reads:

$$d_C(\mathcal{D}', \mathcal{D}) = \min\{d(H', H_i) \mid i = 1, \ldots, m\} \tag{4.7}$$

where H' and H are the color histograms computed from the color component C (see



M_1 :	Car	R_1 G_1 B_1
M_2 :	Glass	R_2 G_2 B_2
M_3 :	Trunk	R_3 G_3 B_3
M_4 :	Leaves	R_4 G_4 B_4
M_5 :	**Track**	**R_5 G_5 B_5**

Figure 4.7: Extraction of color histograms - To compute color histograms, the target object is first searched in the matrix map table. The color of the related matrix is retrieved and used to identify the position of pixels belonging to the object (the track in this example). Finally, the color of the object is read from the frame, at the pixel position determined via the matrix map, and converted into histograms.

Section 4.2.1) of the incoming point cloud $\mathcal{D}'$ and of the point cloud database $\mathcal{D}$ respectively. In Equation 4.7, it is assumed that the database contains m color histograms of the target object, collected from m validated visualizations. The color consistency error is therefore equal to the minimum distance between the color histogram of the target object, in the current frame, and the histograms of the same object extracted across the entire collection of validated frames. The different metrics we considered for comparing histograms are the *Normalized Cross Correlation* [30], the χ^2 (*Chi-square*) [103], the *Intersection* [131] and the *Bhattacharyya* [74]. Such distances, as well as the type of histograms considered will be described in detail in Section 4.6.2.

As stated earlier, histograms are generally found to be good at describing colors in natural images. Such descriptors are relatively invariant to rotation and vary little with respect to scale changes [78]. In our synthetic images, objects undergo various geometric transformations but, thanks to the matrix map and the matrix table, we can always find where the objects are in the scene. Thus, the variance of the histograms to translation and non-affine transformations causes no difficulties in our case. Nevertheless, issues may arise for objects exhibiting complex textures. Textures of non-homogeneous color, present intensity gradients that can be defined at different spatial scales [99]. Since histograms do not account for the spatial organization of color gradients, different textures (i.e. textures with different gradient distributions) may feature similar color histograms. In our approach, colors are assessed through histogram matching (i.e. by computing the distance between histograms), hence, any bug affecting only the gradient organization of textures is unlikely to be detected by our system. To overcome this shortcoming, we split the original object shape into a number of smaller segments before computing the histograms. The purpose of this object segmentation is to reduce big, complex texture patches to smaller patches of possibly more homogeneous color.

We perform segmentation in object space by clustering the incoming point cloud from the current frame according to the centers of the database previously computed (see Algorithm 4.3). This process is shown in Algorithm 4.4. The algorithm produces a matrix, L, of the same size as the current frame. The elements of this matrix indicate (through integers) the various segments (clusters) of the original object. The points from the incoming point cloud P'_o are clustered by computing the minimum Euclidean distances to the database centers (line 3). Indices of such minimum values denote the clusters of the database to which the points P'_o belongs. The matrix L is therefore a *cluster map* which can be used to determine the location of the various segments of the target in the final image. Upon these regions, the color histograms are extracted (Figure 4.8).

4.5 Visual Consistency Error in Screen Space

In the previous sections, we have shown how geometry and color consistency can be assessed through Definition 4.1. In doing so, we have assumed that matrix maps and matrix tables are available during the building of the colored point cloud database as well as when performing bug detection. In this section, we will show that if the rendering pipeline cannot be modified or interrupted during testing, colors can still be assessed in screen space.

To that end, the object space position matrix P_o of the database $\mathcal{D}$ needs first to be first converted to the equivalent screen space position matrix P_s. Equations 4.2 and 4.3 apply here as well, though in reverse order. In effect, this point cloud transformation we seek is what the graphics pipeline normally does for vertices. The object space to screen space conversion of $\mathcal{D}$ is shown in Algorithm 4.5 At its onset, the function TRANSFORMRC performs the *homogeneous divide* (division of the (x, y, z) components by w) through the function NORMALIZE. After this operation, the matrix devCoords will

Algorithm 4.4: Object Segmentation

```
1  function SEGMENT($P'_o$,$P_o$,$M_{wvp}$) returns a matrix of connected components
     inputs:
           matrix $P'_o$ of pixel positions in object space, of size $k \times 4$
           matrix $P_o$ of cluster center coordinates in object space, of size $n \times 4$
           World-View-Projection matrix $M_{wvp}$, of size $4 \times 4$
     locals :
           index array pixelClstIdx of clustered pixels, of size $k$
           index array currPixelClstIdx of pixels belonging to the current cluster
           array fRow of the $Y$ pixels component, of size $k$
           array fCol of the $X$ pixels component, of size $k$
           matrix $L$ of $n$ connected components, of frame size, initially zero
2    for $i \leftarrow 1$ to $k$ do
3        pixelClstIdx[$i$] $\leftarrow$ index of the minimum value of $\|\text{ROW}(P'_o, i) - P_o\|$
4    end
5    for $c \leftarrow 1$ to $n$ do
6        currPixelClstIdx $\leftarrow$ indices $j$ of pixelClstIdx such that pixelClstIdx[$j$] $= c$
7        fRow $\leftarrow P'_o$ (currPixelClstIdx; 2)
8        fCol $\leftarrow P'_o$ (currPixelClstIdx; 1)
9        $L$ (fRow, fCol) $\leftarrow c$
10   end
11   return $L$
```

contain the same points of P_o transformed to Normalized Device Coordinates (NDC) (line 2). Points outside the normalized unit cube (i.e. (x, y, z) points outside the interval $[-1, 1]$) are clipped through the binary mask generated by the function XYZCLIPMASK. The color information about the points too needs to be updated, that is, if the i-th element (row) of P_o has been clipped, the i-th element of C needs to be clipped too. This ensures that the i-th element of both P_o and C will refer to the same pixel. To this end, the color matrix is *masked* with the same index vector cMask used for clipping the device coordinates (lines 4 and 5). The function ROW(X, Y) selects the rows of the matrix X specified by the index vector Y. After applying the viewport transformation (line 6), the variable srcCoords will contain the screen space version of P_o. From the device coordinates, the GPU typically performs *backface culling* [90]. However, as our database $\mathcal{D}$ is made of points rather than polygons, there are no faces to cull but sets of points that model them. This is where the depth buffer Z of the frame comes in useful. Through the function ZTEST, we remove from the matrix scrCoords those points whose z value does not match with the value of the depth buffer at screen position (scrCoords.x,scrCoords.y). The variable zMask is then a binary vector whose

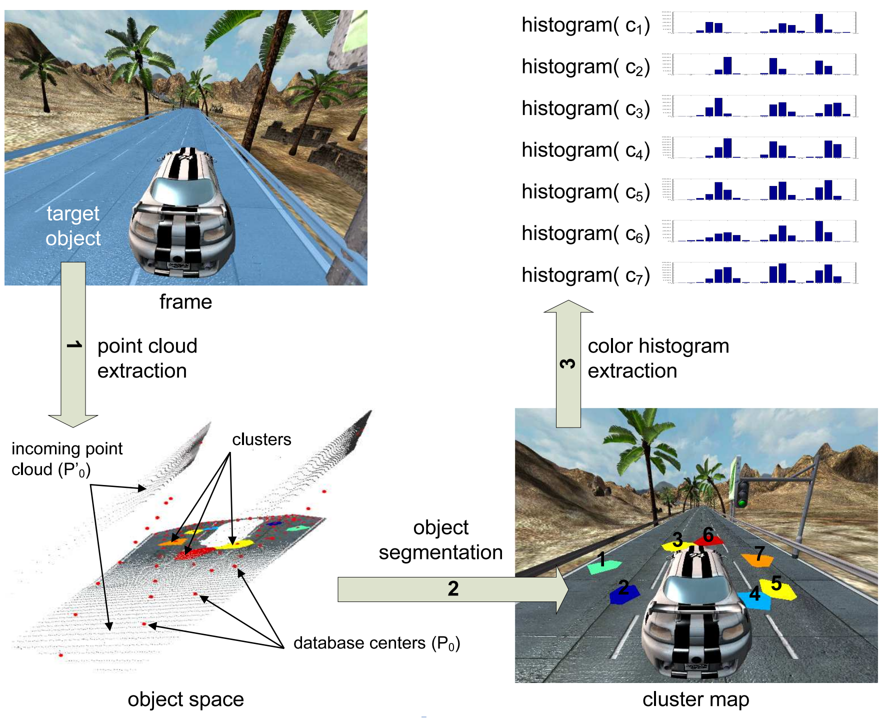


Figure 4.8: Object segmentation - To reduce texture gradient issues, objects are segmented before computing the color histograms. This is achieved by splitting the incoming point cloud of the target object (the track in this example) into clusters whose centers are the points of the point cloud database. The cluster indices of the pixels in P'_o are used to populate the cluster map, which is then used to extract the color histograms from the frame. In this example, the cluster map contains only 7 clusters representing 7 different segments of the target object.

elements are 1 if the relation

$$Z(\text{scrCoords}.x, \text{scrCoords}.y) = \text{scrCoords}.z \tag{4.8}$$

holds; and 0 otherwise. Finally, the screen space colored point cloud $\mathcal{D}_s$ is returned (line 10).

As note earlier, P_o approximates the original object geometry due to the finite resolution of frame and depth buffers. The approximation errors, or *aliasing*, can be observed in the scattered points visible in Figure 4.4. Because of this aliasing, Equation 4.8 needs to be modified so as not to clip too many valid points from the scrCoords matrix. That is, we need to allow for some tolerance τ when performing the z-test. Due to the non-linearity of the depth buffer [90] this tolerance will be a function ZTest of τ and of the pixel depth, which will be read from the depth buffer Z.

Algorithm 4.5: Object space to screen space point cloud conversion

```
1 function TransformRC($\mathcal{D} = (P_o, C), M_{wvp}, M_r, \tau$) returns $\mathcal{D}_s = (P_s, C')$
    inputs:
        matrix $P_o$ denoting the database position matrix, of size $k \times 4$
        matrix $C$ denoting the database color matrix, of size $k \times 3$
        matrix $M_{wvp}$ denoting the world-view-projections matrix, of size $4 \times 4$
        matrix $M_r$ denoting the viewport matrix, of size $4 \times 4$
        matrix $Z$ denoting the depth buffer, of the same size as the current frame
        scalar $\tau$ indicating a tolerance threshold
    locals :
        matrix devCoords of points in Normalized Device Coordinates, of size $k \times 4$
        binary vector cMask denoting a clipping mask, of size $k \times 1$
        vector colorSet storing $(R, G, B)$ color values, of size $k \times 3$
        matrix srcCoords of points in screen space, of size $k \times 4$
        binary vector zMask denoting a clipping mask, of size $k \times 1$
2   devCoords ← Normalize($P_o \times M_{wvp}$)
3   cMask ← XYZClipMask(devCoords)
4   devCoords ← Row(devCoords, cMask)
5   colorSet ← Row($C$, cMask)
6   srcCoords ← devCoords × $M_r$
7   zMask ← ZTest(srcCoords, $Z, \tau$)
8   srcCoords ← Row(srcCoords, zMask)
9   colorSet ← Row(colorSet, zMask)
10  return $\mathcal{D}_s$ = (srcCoords, colorSet)
```

It is important to point out that, besides the database $\mathcal{D}$, the transformation matrix M_{wvp}, the viewport matrix M_r and the depth buffer Z of the new frame also need to be known in order to turn $\mathcal{D}$ into its screen space version. Note, however, that M_{wvp}, M_r and Z can be extracted from the input and output of the rendering pipeline without modifying the rendering process itself.

The purpose of building the screen space database $\mathcal{D}_s$ is to generate a correct or consistent visualization of the target object, for the new frame. In such a way, a screen space comparison can be performed for consistency assessment purposes. Because the position matrix of $\mathcal{D}$ and therefore of $\mathcal{D}_s$ approximates the actual shape of the target object, we should expect the image synthesized through $\mathcal{D}_s$ to be an approximated version, or model, of what we will see in the new frame. For this reason, we shall call the image generated through $\mathcal{D}_s$, the *model frame*.

Due to the approximation errors discussed earlier, the model frame may not be the exact copy of the new frame. A pixel of the new frame at screen position (x, y) coincides with a number of pixels of the screen space database at the same position. These are the pixels contained in the pixel volume depicted in Figure 4.3. The density of the pixel volume depends on the z-distances, in screen space, at which the same region of $\mathcal{D}$ has been observed across the set of validated frames; the bigger such a distance, the sparser the region. The relation between number of pixels and density of $\mathcal{D}$ is illustrated in Figure 4.9. If the number of pixels in the pixel volume is zero, the related pixel in the model frame will be empty. Empty regions in the model frame will be further discussed in Section 4.5.1. Finally, it should be noted that the color of the pixels of the model

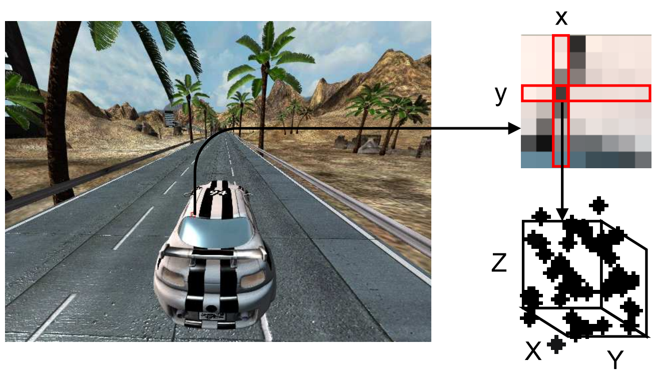


Figure 4.9: Object space vs. screen space pixel volumes - Any screen space position (x, y) has an equivalent region in object space; that is the cube in the picture. The number of pixels of $\mathcal{D}$ lying within the cube are the points that will be projected at (x, y) in the model frame.

frame may not be the same as the color of the pixel of the new frame. This is due to the different light conditions under which the same region of the target object may be observed.

Once we have computed the screen space database $\mathcal{D}_s$ we can measure the visual consistency error $d(\mathcal{D}'_s, \mathcal{D}_s)$ in screen space. Here, $\mathcal{D}'_s$ denotes the position and color of pixels of the target object in screen space relative to a new frame and $\mathcal{D}_s$ the screen space colored point cloud database. In the next section, we will propose an algorithm for measuring the color consistency error in screen space and present some preliminary results. A comprehensive validation of our model based classifier has been performed on the object space approach presented earlier, as this enables the measurement of both geometry and color anomalies. The results of such a validation are presented in later sections.

4.5.1 Color Errors

Before reviewing the algorithm we used for measuring color anomalies in screen space, we wish to clarify the concept of model image density and explain why it is factored into the error measurement process. To that end, we will first explain how a screen space colored point cloud is converted to a (model) frame.

Formally, the screen space cloud is represented through the matrix $\mathcal{D}_s$, whose i-th row is equal to $(\mathbf{p}_i, \mathbf{c}_i)$. Here, $\mathbf{c}_i \in C$ is the color of the pixel at screen position $\mathbf{p}_i \in P_s$. A model frame can then be generated by creating a color image (i.e. a 3D matrix) whose pixels at positions P_s will have colors C. Now, consider the model synthesized frame in Figure 4.10 (right), generated through the process just described. It can be noted that the model frame has some sparse regions compared to the new frame. As

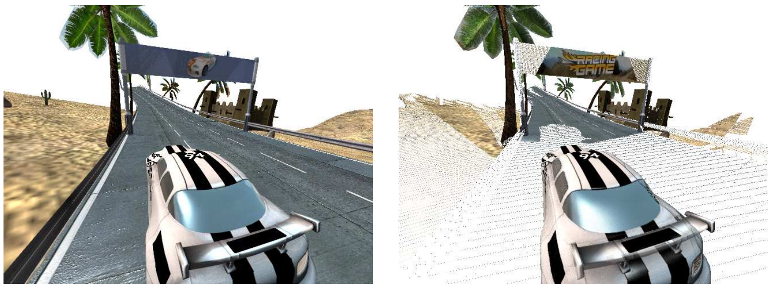


Figure 4.10: New frame and model frame - A model frame is generated through the screen space database $\mathcal{D}_s$, which is built according to the graphics information used by the game to render the new frame.

argued in the previous Section, sparse or empty regions in $\mathcal{D}_s$ may be due to equivalent sparse or empty regions in $\mathcal{D}$. However, an empty region in the model frame can also be caused by an object that does not have a related database $\mathcal{D}$ because it has never been observed in the collection of validated frames. If an object is not present in such a collection we cannot claim that the related empty region is an anomaly as we are not assuming that the validated frames contain all possible objects of the game. Nor should we claim that the region is correct as the information we need for measuring the visual consistency is missing. We can then say that empty regions in the model image are areas of high uncertainty with respect to the visual consistency error that can be measured on them. Finally, sparse regions of the model frame may partially be caused by the aliasing errors introduced in the previous section. Although we allow for some tolerance when computing equation 4.8, if the error exceeds that threshold valid points may be clipped resulting in empty regions of the model frame.

Conversely, a dense region of the model frame is a region that has been observed from the same or a closer distance in the collection of validated frames and/or from different camera angles. Hence, the measure of its visual consistency is expected to be accurate. Dense regions in the model frame are then areas of high certainty. The sparseness of a region in the model can therefore be used as a measure of confidence with which we should accept the visual consistency error computed on the same regions. In this work we used a simple statistical approach to novelty detection similar to the one we reviewed in the previous chapter, Section 3.3.3). In particular, we used the *Multivariate Gaussian Distribution* (MGD) to measure the visual consistency error in screen space, assuming that the color is normally distributed over the surface of the object sub-parts. Instead of performing anomaly detection on a pixel basis, we consider rectangular supports resulting from a *quadtree* subdivision [41] applied to the new frame. Algorithm 4.6 shows the pseudocode of our implementation.

The function ConsistencySS accepts the new frame and the model frame as inputs, among others. The model frame is the one computed from the screen space point cloud, as explained earlier. The algorithm first performs a *quadtree* partitioning of the new colored point cloud A (line 2). A quadtree node is no longer subdivided if the difference between the maximum and the minimum values of each color component in the node is smaller than its respective threshold in τ_p. In our experiments such a threshold was set to $(0.5, 0.5, 0.5)$. We found this to be a good trade-off between speed and accuracy. The output of the partitioning is the set setQ of nodes that make up the quadtree. New and model frames share the same partitioning. For each node, the mean color is extracted through the function Mean which takes, as an input, the (R, G, B) colors of A lying within the node n. The node density, computed by the function Density, is a measure of the sparseness of the model B for the current node, that is, the number of points in B within the node region. Because nodes have been partitioned on the basis of their color homogeneity and not their sparseness, the equivalent region of a node in B may be sparse or even empty. To allow for some tolerance in computing the density of a node, we build a patch of radius ρ centered at the centroid of the current node (line 7). If the equivalent region of the patch in B still does not contain any point, then the node is considered empty and it will be assigned the maximum distance and uncertainty (i.e. ∞) (lines 10 and 11). Otherwise, to find the most appropriate nearby points by which to measure the consistency error we use the *k-Nearest Neighbours* (kNN) algorithm. Specifically, we compute the k nearest model points in the neighbourhood of the node centre nCentre were k is the minimum value between 10 and the set of model points in the patch (line 14). By storing in distO the mean radius of such a set, the variable distO will represent the confidence in making any inference about the node. If the node is dense, there will be enough points in B to compute the visual consistency error and to expect it to be accurate ($\mathsf{distO} = 0$) (lines 17 and 18).

Algorithm 4.6: Consistency Detector in Screen Space

```
1  function ConsistencySS($A, B, \tau_p, \tau_d$,r) returns two gray-scale images
     inputs:
           matrix $A$ denoting the new frame, of size $k \times l$
           matrix $B$ model frame, of size $k \times l$
           vector $\tau_p$ indicating a (R,G,B) partitioning threshold
           scalar $\tau_d$ indicating a density threshold
           scalar $\rho$ denoting a image patch radius
     locals :
           vector mColorN of (R,G,B) values, of size $1 \times 3$
           vector nCentre of screen position coordinates $(x, y)$, of size $1 \times 2$
           matrix patch of screen position coordinates $(x, y)$, of size $r \times r$
           matrix setP of screen position coordinates $(x, y)$, of size $n \times 2$
           matrix vConst of consistency values, of size $k \times l$
           matrix spDist of confidence values, of size $k \times l$
           matrix setO of screen position coordinates $(x, y)$, of size $i \times 2$
           scalar distO denoting a screen space distance
           matrix setC of (R,G,B) values, of size $10 \times 3$
2    setQ ← QTreePartition($A, \tau_p$)
3    foreach node n in setQ do
4        mColorN ← Mean(Color($A, n$))
5        if Density($B, n$) $< \tau_d$ then                // node is sparse
6            nCentre ← Centroid($n$)
7            patch ← Patch(nCentre, $\rho$)
8            setP ← pixels of $B$ within patch
9            if setP is empty then  // no info in the database about node n
10               vConst($n$) ← $\infty$
11               spDist($n$) ← $\infty$
12               continue
13           end
14           setO ← kNN(nCentre, setP, Min($k$, 10))
15           distO ← mean of Euclidean distances from nCentre to setO
16       else                                               // node is dense
17           setO ← pixels of $B$ within node $n$
18           distO ← 0
19       end
20       setC ← kNN(mColorN, Color($B$, setO), Min(Size(Color($B$, setO)), 10))
21       vConst($n$) ← MDG(mColorN, Mean(setC), Covariance(setC))
22       spDist($n$) ← distO
23   end
24   return vConst, spDist
```

Finally, the error is computed with the k colors of the model frame closest to the mean color of the node. The parameter k, again, is set to be the minimum value between 10 and the number of colors of points in B lying either within the current node n or the squared region patch (line 20). From the set of colors extracted, the mean and variance are computed. The consistency error is returned in terms of likelihood, through the multivariate probability density function MDG, of the mean color of the node given the nearby pixels observed in the collection of validated frames (line 21). This process is shown in Figure 4.11 Low probabilities correspond to high inconsistencies. The

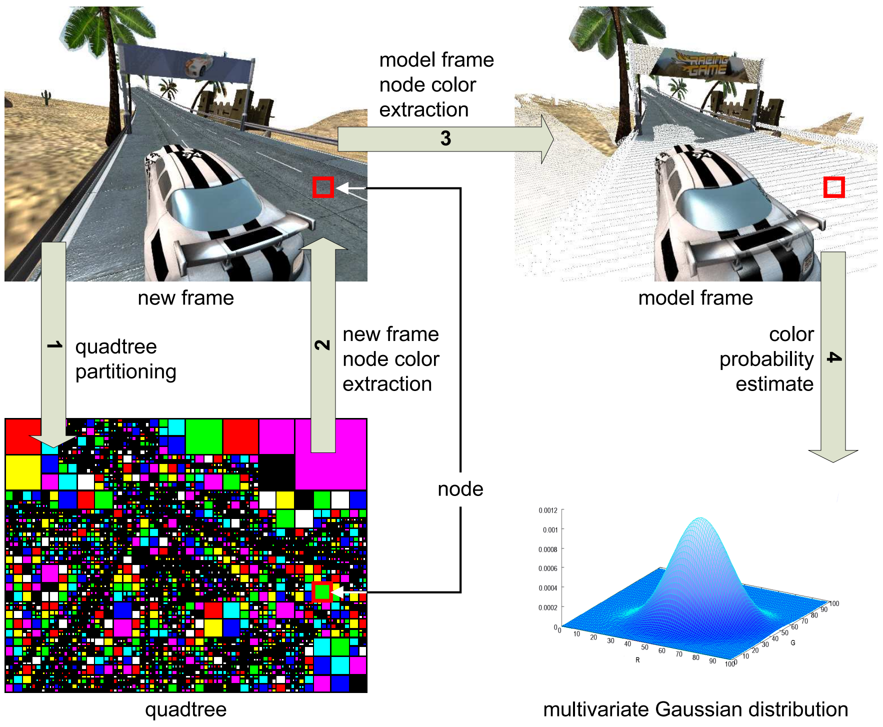


Figure 4.11: Color error in screen space - Color anomalies are expressed in terms of probabilities. As new test frames arrive, they are first partitioned in quadtrees. Then, for each node of the quadtree, the mean color from the new frame is extracted. This value is used to determine the probability value from the multivariate Gaussian distribution, computed from the colors of the node of the model frame.

algorithm generates two images namely, *vConst* encoding the visual consistency error for all nodes of the partitioned frame; and *spDist* representing the confidence about the consistency measured for each node. We call the image vConst the *Visual Consistency Map* and the image spDist the *Confidence Map*. Figures 4.12 (b) and (c) show the output of the algorithm for the test image (a). In order to make the vConst image visually interesting we only displayed those nodes whose consistency error was below

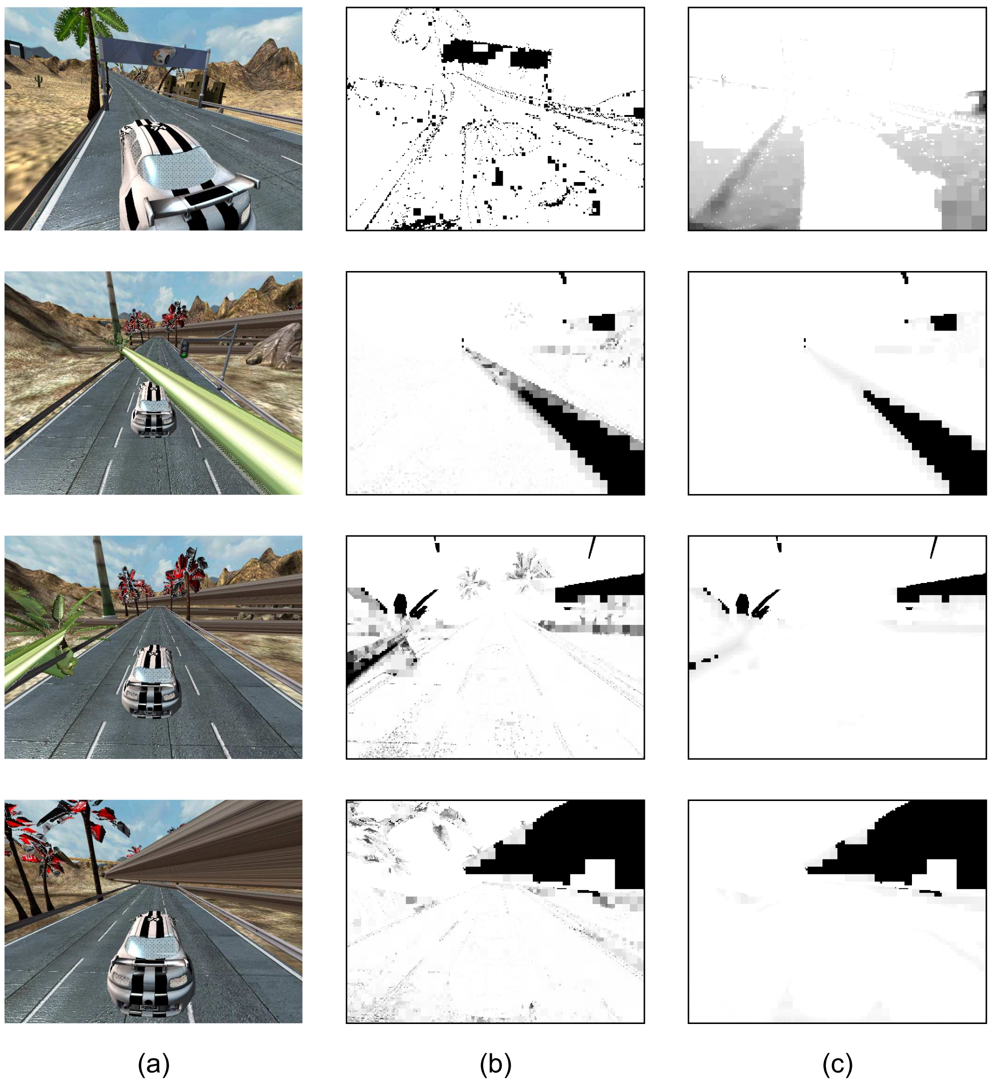


Figure 4.12: Color anomaly detection is screen space - Example of original new frame (a), visual consistency map (b) and confidence map (c). Black regions in the Visual Consistency Map correspond to high inconsistencies with respect to the ground truth database. Dark regions in the Confidence Map correspond to areas of low confidence about the consistency measure.

a certain threshold ($1 \cdot 10^{-6}$ in this case). Areas of high uncertainty (dark regions) in the Confidence Map are located in the correspondence of empty or sparse regions of the model frame. The first row of the Figure contains a bug-free image whereas rows 2, 3 and 4 are images affected by texture corruption (the texture of the palm tree leaves in the background is wrong) and polygon corruption (polygons from the small palm tree and the rocks in the background are wrongly projected). As can be observed, the algorithms correctly detects color anomalies but it also reports high inconsistencies for some consistent pixels in the new frame. For instance, the edges of almost all objects are detected as buggy. This is due to the sparseness of the equivalent region in the database, that is, the number of points in the object space region equivalent to the dark *test pixel* is not enough to determine the correctness of the color, hence the maximum inconsistency reported. The sparseness of a region in object space can be visualized through the Confidence Map (column (c)). Dark pixels represent sparse object space regions and hence, regions upon which consistency should not be measured. Also, note how the banner in in the first row of Figure 4.12 (a) is considered buggy. In this case, the *false alarm* is caused by a texture that was dynamically mapped to the object but never observed in the collection of validated Frames. Finally, it should be noted how the edges of the palm tree, although dense in object space (the related pixels in confidence map are not dark) are still reported as inconsistent. To render such objects, the game used the alpha channel "trick". The apparent complex geometry of the leaves is rendered by using simple meshes mapped with (R, G, B, α) textures. The α channel is used to define transparency. In particular, α is 1 (no transparency) in correspondence of the leaves and 0 (full transparency) elsewhere. The transparent geometry inherits the color of the otherwise hidden background, making the actual geometry look more complex than what actually is. This implies that, for such objects, the point cloud is likely to present high color variance in correspondence of the transparent pixels as the hidden background can potentially be of any color. In such cases, our Gaussian distribution assumption breaks; any color a transparent pixel takes will be detected as improbable.

4.6 Results

In this section, the results from the validation of our algorithms will be presented. In particular, we will show how the performance of model based approach varies with respect to the complexity and size of the target geometries, the type of descriptors and the distance used to match test and database descriptors. The experiments were

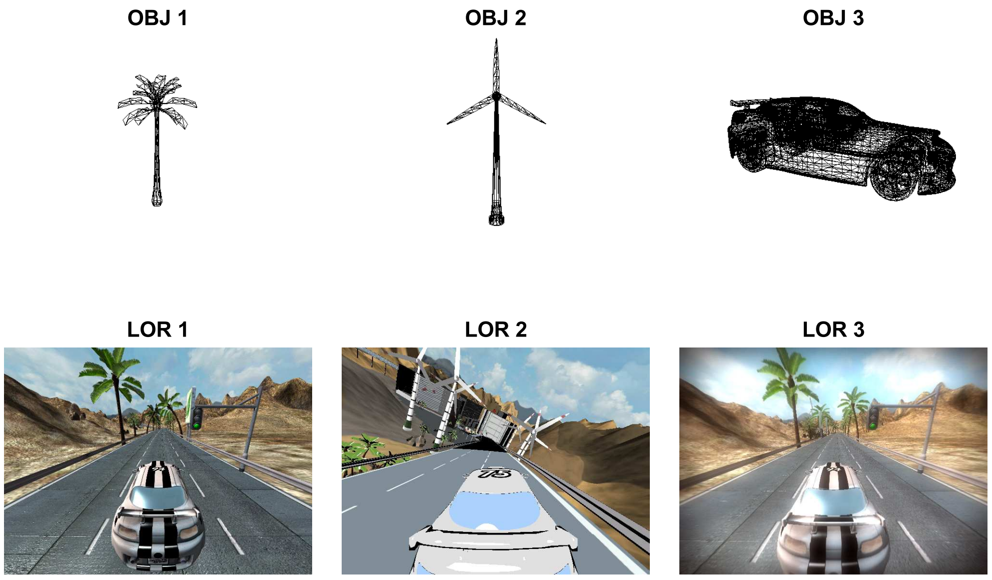


Figure 4.13: Target objects and levels or realism - To assess our model based detectors against geometry and color consistency, different geometries and levels of realisms were used. For geometry tests, a palm tree (OBJ1), a windmill (OBJ2) and a car (OBJ3) where used. Color tests, on the other hand, were performed on three different visualizations of the same environment. These are the basic level of realism (LOR1), the cartoon-like rendering (LOR2) and the nearly photo-realistic environment (LOR3). To achieve different levels of realism, several rendering techniques where used. See Chapter 2 for more details.

performed on three different geometries and three different levels of realism. These are depicted in Figure 4.13.

In order to measure the system accuracy, two test sets were generated; one for geometry measurements and one for color assessments. Both test sets contained a *true set* (i.e. bug-free images) and a *false set* (i.e. buggy images), in approximately the same ratio. The geometry and color bugs of the test set were of random magnitude. In particular, the geometry of the target object was modified by displacing the vertices of the object along their normals at a random distance. The value of such a distance was read from a texture of white Gaussian noise[1]. Similarly, the object color was perturbed by adding a Gaussian white noise to the final frame (Figure 4.14). These anomalies resemble the color and geometry inconsistencies introduced in Chapter 2.

Geometry and color detection accuracy was measured through ROC (Receiver Operating Characteristic) curves. ROC curves are defined on *sensitivity* and *specificity*

[1]In computer graphics, the technique of displacing vertices according to the values sampled from a texture is known as *Displacement Mapping*. Here, a method similar to the displacement mapping technique was used for introducing bugs to the original geometry.

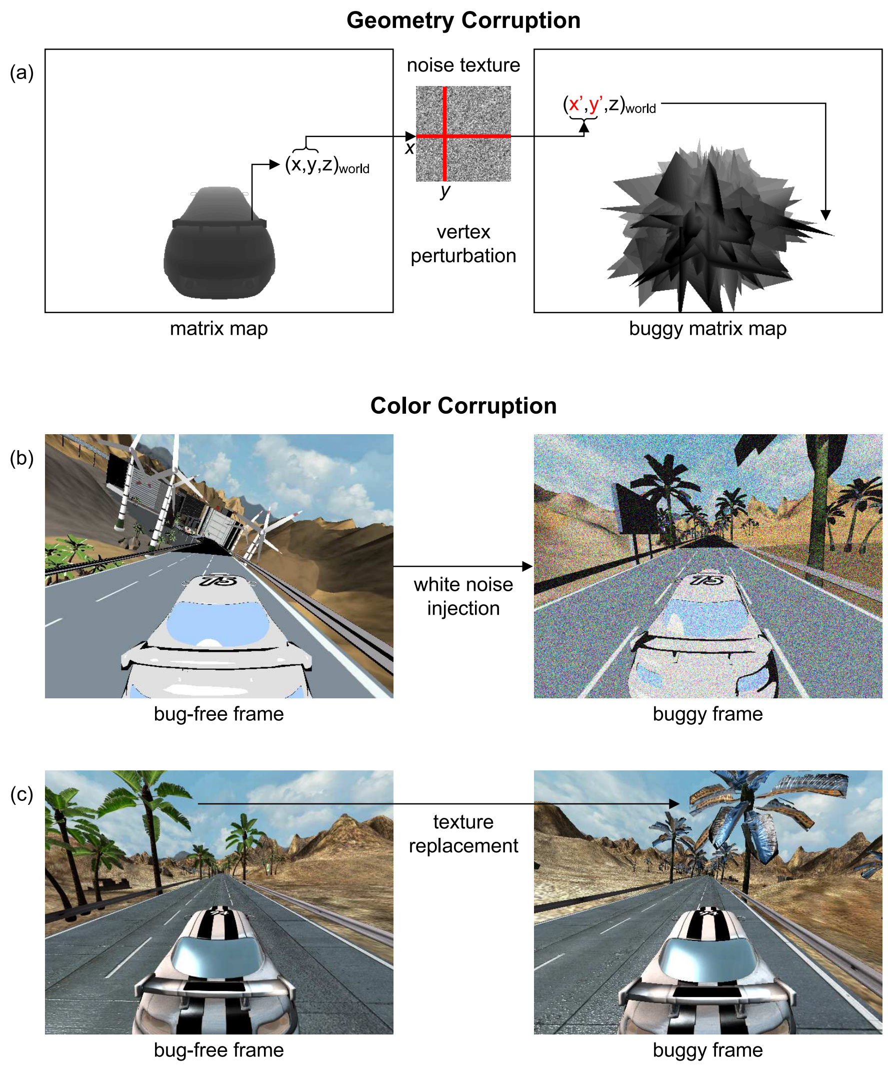


Figure 4.14: Introduction of visual anomalies - In order to reproduce visual anomalies, both the original geometry and color of the object was modified. To generate spike-like bugs, the vertex position of the target object was perturbed through a noise map (a). In particular, the (x, y) value in world coordinates was used as indices to sample the noise texture. The value from the noise texture at (x, y) was used to displace the related vertex, along its normal, of a quantity proportional to the value read from the noise texture. Colors were changed either by adding white Gaussian noise to the frame (b) or by replacing the original texture with a different one (c) (in this example the original texture of the palm tree leaves was replaced with a random texture).

domains, measured as:

$$\mathrm{TPR} = \mathrm{TP}/(\mathrm{TP} + \mathrm{FN}) \qquad \text{sensitivity} \qquad (4.9)$$

$$\mathrm{FPR} = \mathrm{FP}/(\mathrm{FP} + \mathrm{TN}) \qquad 1 - \text{specificity} \qquad (4.10)$$

where TP is the number of true positives (hits), FP is the number of false positives (false alarms), TN the number of true negatives (correct rejections) and FN the number of false negatives (misses). False positive rate (FPR) and true positive rate (TPR) are the domain and co-domain of the ROC curve respectively. To measure accuracy, we computed the Area Under Curve (AUC), which is the integral of the ROC curve over the false positive rate. False and true positive rates were computed on true sets and a false set. Figure 4.15 depicts examples of ROC curves computed for three target objects and three environments.

All colored point clouds were built via modifying the target game engine, as explained in Section 4.3. The database point cloud was built on a *training* set, that is, a set of validated frames. Target point clouds were extracted from *test* sets, composed of a positive set and a negative set. Both training and test sets resulted in a collection of frames, matrix maps and matrix tables. These latter were stored in *xml* files to facilitate their access.

4.6.1 Geometry Tests

Geometry consistency was measured by using Equation 4.6. Accordingly, the target geometry was considered inconsistent if the Hausdorff distance between the target point cloud and the database point cloud was bigger than a threshold. In order to build ROC curves, such a threshold was varied between the minimum and the maximum distance measured across the test set. The point cloud database underwent an adaptive clustering (see Algorithm 4.3); this enabled us to reduced the number of points in the point cloud, and therefore memory and computing complexity. Adaptive clustering was performed using four cluster radii, of size 15, 30, 60 and 120. The results of the geometry test are shown in Figure 4.16.

Results are displayed through box plots and decision trees [17]. As it can be noted from the box plots, the median (denoted by red segments) is very high (close to 1) for almost all test cases. Also, note that the *inter quartile range* (the height of the boxes) is very small for all boxes. This indicates that accuracy does not greatly vary around the median value, throughout the test set. These results indicate that the system is highly robust and accurate. In order to determine how the cluster radius influences

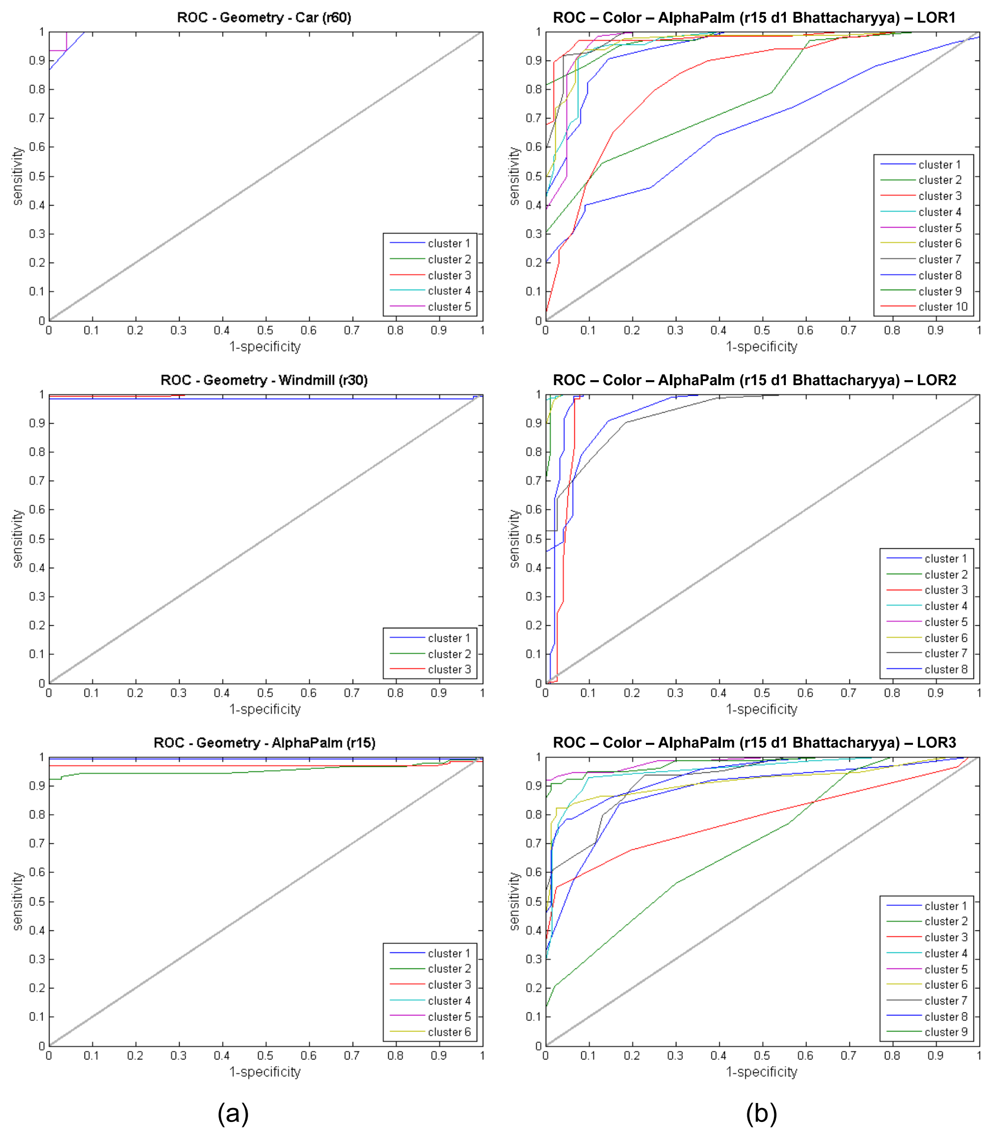


Figure 4.15: ROC curves from the model based classifiers - ROC curves were used to measure the effectiveness of the detectors, in terms of false positive (1 - specificity) versus true positive ratio (sensitivity). A system performs best if it never reports either false positives or false negatives. In these cases, the area under the ROC curve (AUC) is at its maximum (i.e. 1). Poorly performing systems produce ROC curves whose area approaches 0.5. Low accuracy curves are the ones lying close to the straight, diagonal gray line in the graphs. The graphs show (a) the geometry results for three different objects (Car, Windmill and AlphaPalm) and (b) the color results for the same object (AlphaPalm), rendered using three different level of realism (LOR1, LOR2 and LOR3). Different curves within the same graph represent different clusters (sub parts) in which the original geometry was segmented. Because sub parts may have different geometrical complexity and size, performance may vary amongst them. The graphs title contains information about the radius, descriptor and distance used. For example, the information *(r15 d1 Bhattacharyya)* indicates that the system has been trained and tested with radius 15, color descriptor 1 and histogram distance *Bhattacharyya*.

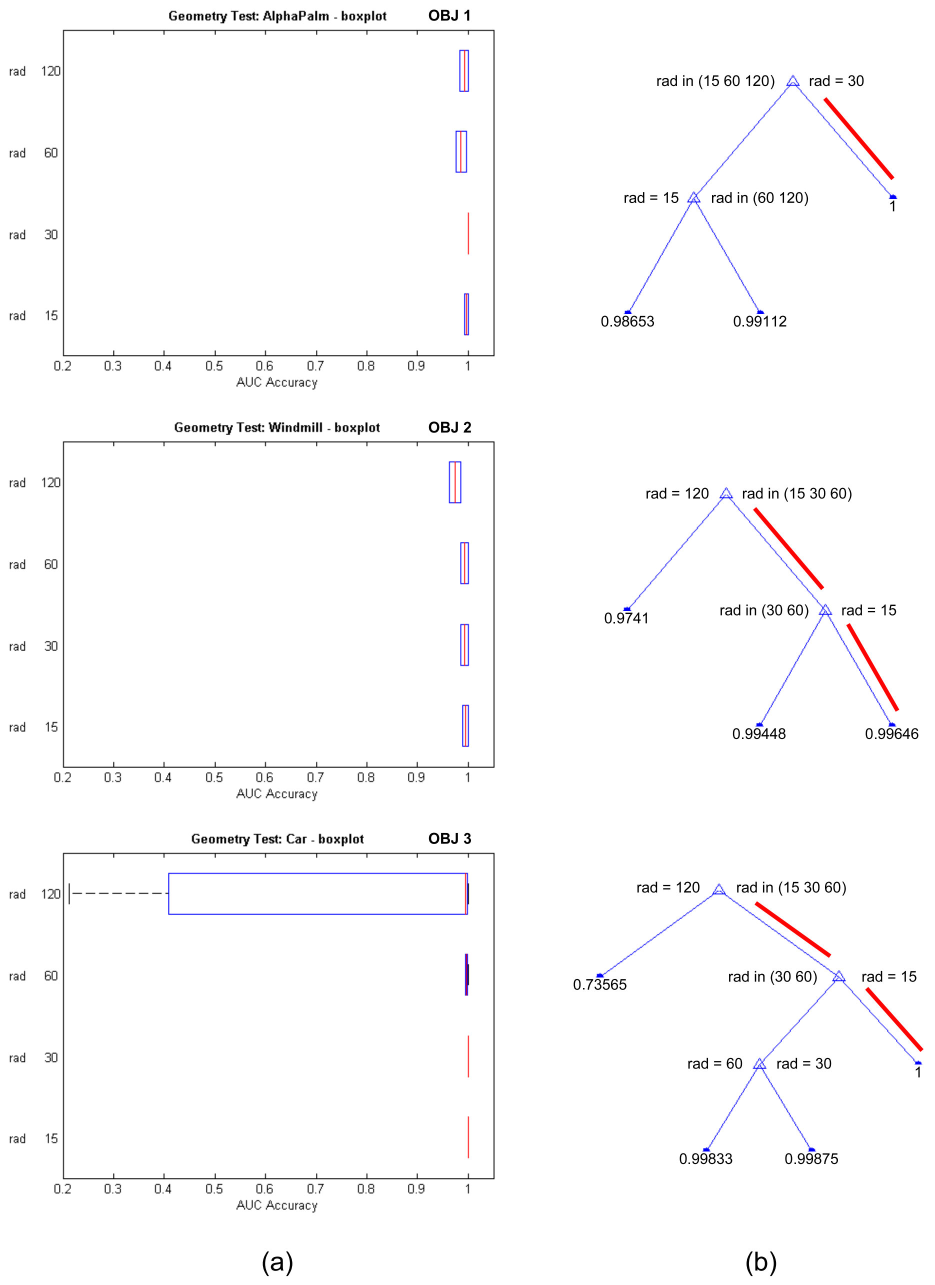


Figure 4.16: Geometry tests - To validate the system upon geometry issues, three different objects where tested: a palm tree, a car and a windmill. For each one of such objects, true and false sets were generated to measure the accuracy via AUC (area under curve). Tests were performed considering cluster radii of 15, 30, 60 and 120. The box plots in column (a) show the results from our tests. For all radii considered, the system was highly accurate (around 99%). To determine how the cluster radius influenced accuracy, we used regression trees. As it can be observed from column (b). A cluster radius of 15 is typically the optimal choice as it enables the system to be 100% accurate, for 2 out of 3 objects analyzed. Conversely, a cluster radius of 120 was clustered by the decision trees as worst choice. The red segments along the branches of the regression trees indicate the paths of maximum accuracy.

the system performance, we used two statistics techniques. These being *Decision Trees* [17] and the *Analysis of Variance* (ANOVA) [62].

Decision trees, are often used for predicting the response as a function of predictors. In our case, the predictors are the cluster radii; the response is the accuracy, measured as Area Under Curve (AUC). Decision trees are binary trees where each branching node is split based on the values of the observation (cluster radii and related accuracy). Decision trees are often *pruned* in order to reduce the complexity of the final classifier as well as to endow the classifier with better predictive power. Pruning decision trees consists in removing branches that give least improvement in error cost. A level 0 pruning equals no pruning; a maximum level pruning produces a tree with only one node — the root node. All trees where pruned at the $n-2$ level, where n was the maximum pruning level. This allowed us to visualize the factors that best explained the variance of the accuracy; thus facilitating the interpretation of our results. As it can be noted from Figure 4.16 (b), a common pattern emerges from the classification. All trees determine that radius 120 produces lower accuracy values, in general. Likewise, a cluster radius of 15 is classified as a high accuracy factor. From these observations, it would be tempting to conclude that smaller cluster radii make geometry classifiers more accurate. However, there is no guarantee that the accuracy based discrimination between factors (e.g. cluster radii) produced by the decision trees is statistically significant. In fact, this seems not to be the case; the medians of the box plots are all close to one another, regardless of the cluster radius considered.

To determine whether the aforementioned discrimination is statistically significant, we used the 1-way ANOVA test. Through the ANOVA, we computed the p-values (the statistical significance) for the *null hypotheses* on the main factors. Our null hypothesis stated that there is no radius-sample mean that is significantly different from any other radius-sample mean. Under such an hypothesis, the mean accuracy produced by cluster radius 15 is not significantly different from the mean accuracy produced by cluster radii 30, 60 and 120. Similarly, the mean accuracy produced by cluster radius 30 is not significantly different from the accuracy mean produced by cluster radii 60 and 120 and so on. A small p value (e.g. ≤ 0.01) indicates that differences between effects means (mean accuracy upon different cluster radii) are highly significant. The probability of this outcome under the null hypothesis is equal to the p value. Therefore, upon small p values it is safe to reject the null hypothesis. The geometry results from the ANOVA tests are reported in Table 4.2.

The standard ANOVA table has six columns, these are the source of the variability (Source); the sum of squares (Sum Sq.) due to each source; the degrees of freedom (d.f.) associated with each source; the mean squares (Mean Sq.) for the source; the F-statistic, which is the total variation between and within samples; the p value (Prob $>$ F), which is the probability of the observation under the null hypothesis, that is, all observations coming from populations (radii) with the same mean. For more details on these parameters the reader is referred to [62]. The columns in gray in the table denote the p values. We reject the null hypothesis if the p-value is less or equal to 0.01 (significance level)[1]. Values displayed in bold in the table indicate statistically significant factors ($p \leq 0.01$). Because the p values are big for all objects we conclude that, insofar as what geometry tests are concerned, the (high) performance of the system does not depend on the choice of the cluster radius.

	Hue	**Saturation**	**Value**
1	10	0	0
2	12	0	0
3	14	0	0
4	16	0	0
5	18	0	0
6	20	0	0
7	8	2	0
8	9	3	0
9	10	4	0
10	11	5	0
11	16	6	0
12	13	7	0
13	6	2	2
14	7	3	2
15	8	4	3
16	9	5	5
17	10	4	4

Figure 4.17: Color descriptors - Color tests were performed on 17 histograms of Hue, Saturation and Value components. The values of the elements in the table indicate the number of bins used for the related component. For example, histogram 13 is composed by 6 Hue bins, 2 Saturation bins and 2 Value bins.

4.6.2 Color Tests

Color consistency was assessed through 17 different color histograms defined in HSV space (i.e. Hue, Saturation and Color). To this end, RGB colors from the target object were first transformed to HSV colors. Then, HSV colors were *binned* into

[1]In statistics, common significance levels are 0.05 or 0.01 [62].

the histograms according to the histogram type. Figure 4.17 illustrates the types of histograms used in this work. The similarity between database and test histograms was performed according to Equation 4.7, that is, the distance between target histogram and database histograms was computed and its minimum value considered. In this work, we explored four different metrics for histogram matching, formulated as:

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - \overline{H}_1)(H_2(I) - \overline{H}_2)}{\sqrt{\sum_I (H_1(I) - \overline{H}_1)^2 \sum_I (H_2(I) - \overline{H}_2)^2}} \tag{4.11}$$

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I) + H_2(I)} \tag{4.12}$$

$$d(H_1, H_2) = \sum_I \min(H_1(I), H_2(I)) \tag{4.13}$$

$$d(H_1, H_2) = \sqrt{1 - \frac{1}{\sqrt{\overline{H}_1 \overline{H}_2 N^2}} \sum_I \sqrt{H_1(I) H_2(I)}} \tag{4.14}$$

These are the *Normalized Cross Correlation* (Equation 4.11), χ^2 (Equation 4.12), *Intersection* (Equation 4.13) and *Bhattacharyya* (Equation 4.14) distances. In the above set of equations, $H_k(I)$ represents the I-th bin of the k-th histogram and

$$\overline{H}_k = \frac{1}{N} \sum_J H_k(J) \tag{4.15}$$

where N is the total number of histogram bins.

As for the geometry tests, the color information concerned sub-geometries or segments computed as shown in Algorithm 4.3, with cluster radii 15, 30, 60 and 120. The purpose of segmenting the original shape of the object was to reduce the texture complexity of the original object, as explained in Section 4.4.2. We validated our system on one complex object (a palm tree) across three different levels of realism (Figure 4.13). These are:

- Level of Realism 1 (LOR1): diffuse, ambient and specular light contributions
- Level of Realism 2 (LOR2): sharp threshold on diffuse, ambient and specular light components
- Level of Realism 3 (LOR3): diffuse, ambient, specular light contributions; motion blur and bloom

Figure 4.18 shows the outcome from the color test performed. As for the geometry

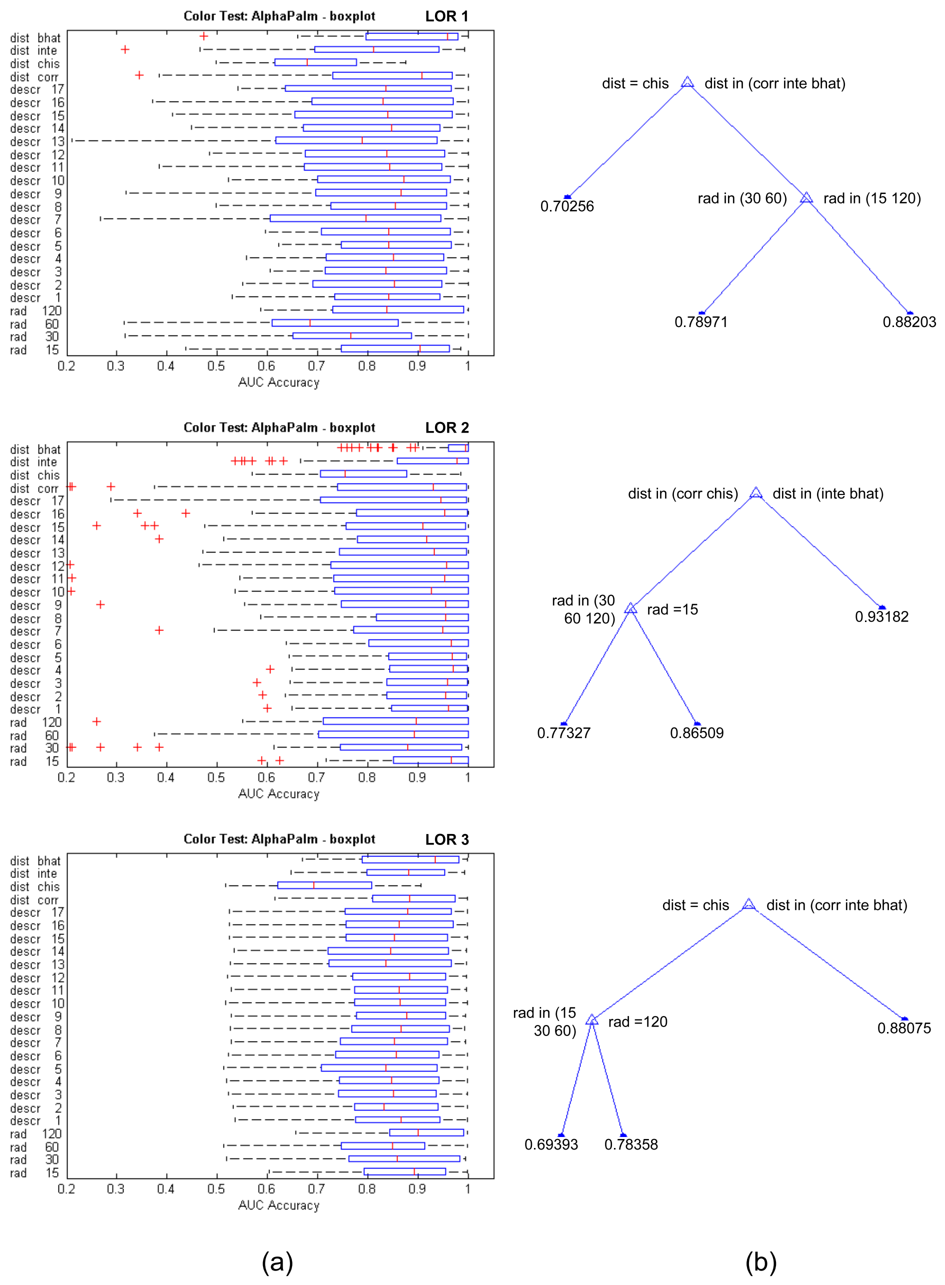


Figure 4.18: Color tests - Color tests were performed on the object (AlphaPalm), rendered with different techniques or levels of realism. The first row refers to level of realism 1 (LOR 1); the second row to LOR 2 and the third row to LOR 3. The parameters of the system where the cluster radius (rad) of values 15, 30, 60 and 120; the color descriptor (descr) taking values from 1 to 17; and the histogram distance, that is, Correlation (corr), χ^2 (chis), Intersection (inte) and Bhattacharyya (bhat). The box plots (a) computed on the test data, show how each parameter influences the final accuracy of the system. The decision trees (b) predict the accuracy value as a function of the parameters. In this example, trees are pruned at the $n - 2$ level, where n is the maximum pruning level.

tests, box plots and decision trees were used to facilitate the interpretation of the results. However, compared to the geometry tests, the amount of test cases was much larger for color assessments, for two additional parameters needed to be accounted for, besides the cluster radius. These were the color descriptor and the histogram distance. The color descriptor took values from 1 to 17; the histogram distance took values from the domain *Normalized Cross Correlation*, χ^2, *Intersection* and *Bhattacharyya*. The box plots show the influence of each one on the aforementioned parameters to the final system accuracy. As with geometry tests, the decision trees predict the system accuracy as a function of the parameters.

By looking at the box plots, it can be observed that the Bhattacharyya, the Intersection and the Correlation distances produce more accurate results, compared to the χ^2 distance. This result is confirmed by the decision trees, which tend to classify χ^2 as a variable that produces lower accuracy. The decision trees also show that the choice about the cluster radius is subject to the level of realism considered. For nearly photo-realistic environments (LOR1), radii of 15 or 120 represent an optimal choice, producing systems which are 88% accurate, on the average. For cartoon-like rederings (LOR2) or when post-processing effects are used (LOR3), the cluster radius does not seem to play a major role in the sytem accuracy, if a good distance is used. In such cases, the Bhattacharyya or the Intersection distances can make the system about 90% accurate, regardless of the radius used. In general, however, cluster radii of 15 and 120 seem to perform best.

As for geometry tests, we performed the ANOVA test in order to measure the statistical significance for the *null hypotheses* on the main factors. In fact, besides being interested in computing the p-values just for the main effects (i.e. radius, descriptor and distance), we now also wish to test whether the accuracy is related to multiple factors (grouping variables). For example, we want to know if the accuracy depends on the radius as well as the descriptor; or on the radius and the distance; or on the radius, the distance and the descriptor, considered jointly; and so on and so forth. This type of test it is called N-way ANOVA [62]. As with the 1-way ANOVA test, small p values will cast doubt on the associated null hypothesis.

The results from the N-way ANOVA test are reported in Table 4.3. The gray column is relative to the p values. Values displayed in bold indicate statistically significant factors ($p \leq 0.01$). In accordance with our observations from the box plots and decision trees, we note that the accuracy depends significantly on the choice of the radius and the distance, for all environments considered (i.e. LOR1, LOR2 and LOR3). We also note that cluster radius and distance measure, together, play an important role in the effectiveness of the system, as far as LOR2 is concerned. The color descriptor becomes discriminative only for LOR2, that is, the cartoon-like environment. This result can be observed in the middle box plot of Figure 4.18 (a). Indeed, note that descriptors 1 to 8 — the ones containing the Hue component mostly — perform better than others, for the cartoon-like environment (LOR2). Such descriptors have high median and small inter quartile range, compared to the rest of the descriptors.

It is worth noting that the inter quartile ranges of the color box plots are typically bigger than those of the geometry box plots. Likewise, the medians are generally lower. This indicates that the system is not as accurate at detecting color inconsistencies as the geometry detector. Such a result should not be surprising, however. As mentioned at the onset of this chapter, the object geometry does not change in object space, regardless of the environment used. Because the geometry error is computed in object space, the accuracy is expected to be high. Conversely, in photo-realistic environments, the color over the object surface is influenced by many factors and does not have an environment-invariant representation. Nevertheless, very effective model based color detectors can be built upon a careful selection of the system parameters. Table 4.1 reports the parameters of the best performing geometry and color detectors, for the objects and the levels of realism considered in this work. The values reported in the table have been determined by considering the best performing branch of the aforementioned decision trees. In particular, the best accuracy value for each object or rendering level is the one corresponding to the leaf of the rightmost branch of the related non-pruned tree (see Figure 4.16 and 4.18 (b) for a reference). Best radii, descriptors and clusters correspond to the parent nodes of such a leaf. For example, the best accuracy leaf for LOR 3 (95.88%) was the one having the nodes $\{\text{Distance} = \textit{Bhattacharyya}\}$, $\{\text{Descriptor} \in \{9, 10, 11, 12\}\}$ and $\{\text{Radius} = 120\}$ as ancestors. Note that the table contains multiple values for some parameters (e.g. the best system for LOR 2 can be built with color descriptors 9, 10, 11 and 12). Such values have been clustered under the same node by the decision tree and therefore produced similar accuracy values, given the rest of the nodes. Finally, note that the only parameter to tune for building accurate model based geometry classifiers is the cluster radius. Geometries are described through point clouds; the distance between a test geometry and the related database point cloud is computed through the Hausdorff metric, as explained earlier.

When multiple radius choices are available (as for LOR 2 in the table) the biggest values (i.e. 120) should be selected for building the classifier. Bigger cluster radii produce smaller number of clusters or sub-geometries (Figure 4.19). A smaller number of clusters entails a smaller number to elements to process and store, thus less

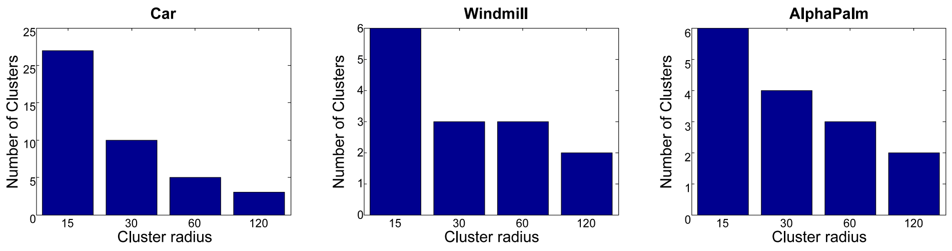


Figure 4.19: Number of clusters versus cluster radius - The smaller the cluster radius, the larger the number of parts into which the original geometry will be split. The graphs refer to three different geometries, i.e. a car, a windmill and a palm tree.

computational and memory loads.

Table 4.1: Best Performing Model Based Classifiers

Geometry			Color				
Object	Radius	Accuracy (AUC)	Environment	Radius	Descriptor	Distance	Accuracy (AUC)
AlphaPalm	30	1.00000	LOR 1	120	16	*Bhattacharyya*	0.95211
Windmill	15	1.00000	LOR 2	60,120	2	*Bhattacharyya*	0.99516
Car	15	0.99646	LOR 3	120	9,10,11,12	*Bhattacharyya*	0.95881

4.7 Summary

In this chapter we have shown that visual consistency can be quantitatively measured for both geometry and color assessment purposes. Such a measure can be highly effective if we assume to have matrix maps and matrix tables available for building both training and test sets. If the graphics pipeline cannot be modified or interrupted during testing, however, the consistency error can no longer be computed in object space. We have suggested a way to overcome this difficulty in probabilistic terms, by measuring the consistency error in screen space directly. To that end, we employed a Multivariate Gaussian Distribution to express the color probability of pixel regions. This approach turned out to be effective for non-photorealistic environments. For more complex environments, however, more complex distributions are required (e.g. Multivariate, Gaussian Mixture models).

In validating the model based classifiers, we noted that the two most important parameters to tweak are the cluster radius and histogram distance. In fact, as far as geometry assessments are cons, the only parameter to determine is the cluster radius. Using an appropriate parameter set, the model based detector can reach very high accuracy levels — between 95% and 99%. Color inconsistency detectors performed slightly worse, most often when testing the highest quality environments. Color classifiers store the color histograms from the target objects of the training set and use them later to match test color histograms. In the case of photo-realistic environments, however, objects are hardly ever rendered with the exact same colors more than once. Rather, in such environments the color of the surface of an object depends on a number of environment factors, such as the color of the scene lights, the color of nearby objects and the position and pose of the object with respect to the camera.

By contrast, the object space representation builds arbitrarily accurate point clouds to model the original consistent geometry. Given that the shape of virtual objects never changes in object space, anomalous points will necessarily lie far from the consistent point cloud. Such anomalies can be effectively identified through the Hausdorff metric.

Table 4.2: Model Based Classifiers - ANOVA - Geometry Tests

	AlphaPalm (OBJ 1)					Windmill (OBJ 2)					Car (OBJ 3)				
Source	Sum Sq.	d.f.	Mean Sq.	F	Prob > F	Sum Sq.	d.f.	Mean Sq.	F	Prob > F	Sum Sq.	d.f.	Mean Sq.	F	Prob > F
radii	0.00023	3	7.52e-005	0.84	0.54	0.00056	3	0.00019	1.45	0.3544	0.15615	3	0.05205	1.01	0.4356
Error	0.00036	4	8.99e-005			0.00052	4	0.00013			0.41056	8	0.05132		
Total	0.00059	7				0.00109	7				0.56671	11			

Table 4.3: Model Based Classifiers - ANOVA - Color Tests

	Level of Realism 1 (LOR 1)					Level of Realism 2 (LOR 2)					Level of Realism 3 (LOR 3)				
Source	Sum Sq.	d.f.	Mean Sq.	F	Prob > F	Sum Sq.	d.f.	Mean Sq.	F	Prob > F	Sum Sq.	d.f.	Mean Sq.	F	Prob > F
rad	1.9056	3	0.63520	28.73	**0**	1.5241	3	0.50803	20.78	**0**	0.5253	3	0.17509	14.28	**0**
descr	0.5229	16	0.03268	1.48	0.1	1.1599	16	0.07249	2.97	**0.001**	0.0849	16	0.00531	0.43	0.9743
dist	3.5343	3	1.17809	53.29	**0**	4.1511	3	1.38370	56.60	**0**	5.2993	3	1.76644	144.05	**0**
rad*descr	0.2333	48	0.00486	0.22	1	0.4125	48	0.00859	0.35	1	0.0535	48	0.00111	0.09	1
rad*dist	0.2711	9	0.03012	1.36	0.2006	0.6047	9	0.06719	2.75	**0.0036**	0.0792	9	0.00880	0.72	0.6928
descr*dist	0.1991	48	0.00415	0.19	1	1.0095	48	0.02103	0.86	0.7385	0.1731	48	0.00361	0.29	1
rad*descr*dist	0.2943	144	0.00204	0.09	1	0.3438	144	0.00239	0.10	1	0.0662	144	0.00046	0.04	1
Error	22.5514	1020	0.02211			21.6126	884	0.02445			12.5080	1020	0.01226		
Total	32.0540	1291				30.8751	1155				21.3410	1291			

5

View Based Classifiers

As we have seen in the previous chapter, it is possible, by reversing the graphics pipeline, to bring the geometry and color information from screen space back to object space and use it to make some useful inference. We called such an approach *model based classification*, for the discrimination between bugs and non-bugs relies solely on the acquired knowledge (of shape and color) about individual models of the virtual objects. This knowledge is not conditioned to where the objects are in the world and where the camera is looking. Since the color over the surface of the objects can, in fact, vary with both the object and the camera position — among other factors — the model based approach, in such cases, is liable to make inaccurate predictions. In order to make robust inference in photo-realistic environments, we further develop our approach by adding camera-object interactions to our model, to form a *view based* classifier. We postulate that allowing for this additional component will improve the detection of color anomalies in 3D virtual environments.

5.1 General Approach

The camera-object interaction, as well as its visual outcome from the rendering system, is encoded in the behaviour of the virtual environment. However, in line with the assumptions we made when building the model based classifier, all we know about the rendering system is a collection of validated (bug-free) frames. The consistent behaviour of the system itself is not explicitly given. The approach used in this chapter elicits knowledge of the camera-object interaction rather than the objects alone. The research question we pose is:

How can a mechanism "learn" the relationship between the scene camera *and single object appearance and use it to detect visual inconsistencies?*

The problem of learning and prediction is ubiquitous in machine learning research and, indeed, numerous techniques to address these issues have been developed and refined over the years. In this work, we will consider the two categories within which all machine learning strategies fall, namely, *regression* and *clustering*.

Regression aims at eliciting a mapping between two feature spaces. That is, given an input feature space, X in $\mathbb{R}^n$, and an output space, Yin $\mathbb{R}^m$, regression algorithms try to find an *estimator* which transforms X into Y. In our case, X is the feature space in which camera-object interactions are defined, and Y is a space denoting the geometry and color appearance of our virtual entities. The estimator is built in order to predict the object appearance for debugging purposes. Clustering, on the other hand, consists of grouping data or observations of a feature space X into subsets (clusters), based only on the information that can be inferred from the data. Observations in the same cluster are thus similar in some sense. In our work, we aim at forming clusters of consistent rendering system inputs and outputs defined on some feature space. The feature space will need to encode some interesting information about the camera-object interaction and object appearance. If the feature space is appropriately chosen, then inconsistent interactions and/or visualizations will lie far from the clusters of normal or consistent data instances. Hence, anomaly detection will come down to measuring the distance of a new vector to the nearest consistent cluster. Large distances will indicate suspicious camera-object interactions and/or visualizations.

Either through regression or clustering, once the mapping between camera-object information and object appearance has been suitably represented, it can be used to make inferences about new stimuli from the system to test. By computing the similarity between the prediction from our testing mechanism and the actual data from the game, we will be able to quantify the consistency of the appearance (Figure 5.1). Building and using estimators and clusters for debugging purposes is the subject of subsequent sections.

5.2 Appearance Description

It is an accepted fact that the effectiveness and robustness of any classifier, whether estimator or cluster based, strongly relies on the nature of the input data, described as a set of attributes or descriptors [23]. As our ultimate aim is to build classifiers of consistency, we will need to turn our data — the relationship between the camera

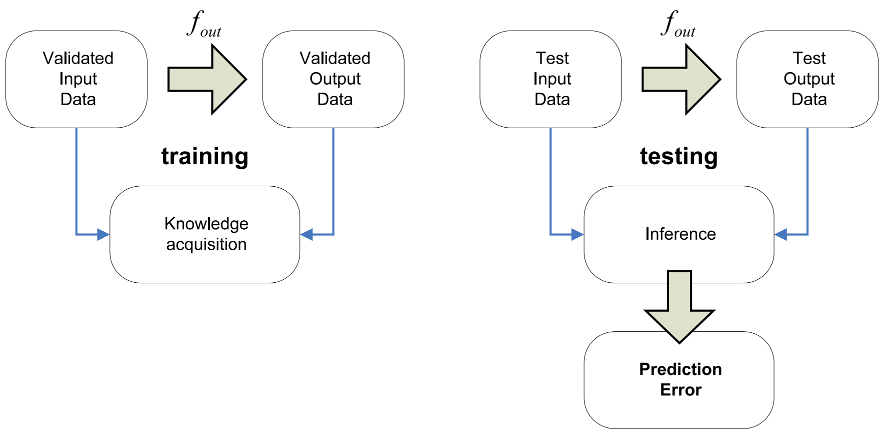


Figure 5.1: View based consistency estimation scheme - In order to acquire the target behaviour of the rendering system, valid input and output samples needs to be collected. The behaviour can then be modeled, for example, through machine learning mechanisms. This schematic diagram on the left side of the figure represents this phase. The knowledge acquired can be used to assess the system behaviour for debugging purposes. On the right hand side of the figure, a diagram shows the test phase, in which inference is made by the mechanism upon new stimuli received from the system to test.

and the appearance of the object on screen — into descriptors to use for training our estimators and building our clusters. In doing so, attention must be paid to control the complexity of the descriptors. Indeed, empirical evidence shows that the dimensionality of descriptors is a critical factor for both classification and regression problems [130]. Highly dimensional feature spaces produce, in general, badly performing inference mechanisms. Therefore, not only should our descriptors be discriminative, but they should also be of limited dimensionality.

5.2.1 Geometry and Color Descriptors

Since we aim at describing the (2D) appearance of objects on screen, geometry descriptors should ultimately represent the 2D shape of objects. We decided to represent the object silhouette through seven shape descriptors. These are the object area (in pixels), the distance from the camera $\in [0, 1]$, the coordinates of the silhouette centroid $\in [0, 1]$, the length of the major and minor axes of the ellipse encompassing the object silhouette, and the angle between the x-axis and the major axis of the aforementioned ellipse $\in [-90°, 90°]$. A vector of any combination of such descriptors shall be referred to as a *geometry descriptor*. The geometry descriptors can therefore keep track of the object size, its distance from the camera, its eccentricity (ellipse axes) and orientation (ellipse orientation) (Figure 5.2 left). We decided not to use any affine invariant 2D shape descriptor such as the affine invariant Fourier descriptor (AIFD) [22] the CSS descriptor (Curvature Scale Space) [102] or the generalized Hough transform descriptor (GHTD) [8]. The aim of such popular feature vectors is to extract some shape *signature* that is invariant to affine translation, rotation and scaling. However, because we want to correlate the shape and position change to the camera-object interaction, the invariance properties of such representations cannot be exploited to serve our purposes. Moreover, most of these methods are *boundary based* (i.e. they only rely on the information about the contour of the object) and, as such, are strongly affected by shape irregularities due to occlusions [104]. We postulate that our geometry descriptors — being *region based* — are more suitable to fulfill our aims, for the objects and environments considered feature occlusion and cluttering. Finally, the geometry descriptors we propose can be computed cheaply, as the most expensive operation is the extraction of the second moments, which can be efficiently calculated [104].

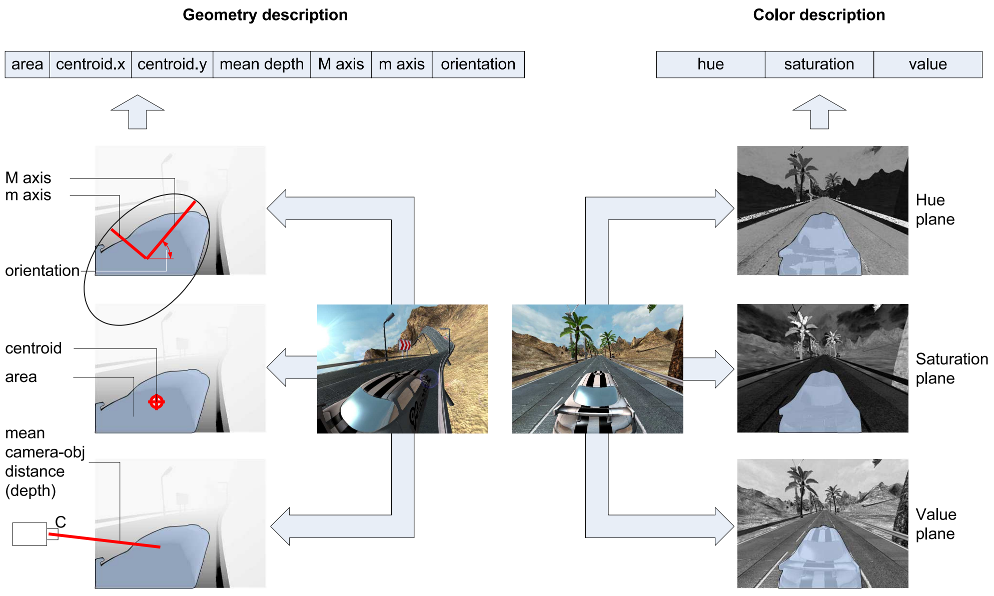


Figure 5.2: Geometry and color descriptors of appearance - The shape and color of objects on screen is described using vectors of specially selected components. Geometry descriptors allow for object area (in pixels), centroid position and other shape properties. Colors are defined through histograms of Hue, Saturation and Value planes extracted from the image.

The color distribution is described using color histograms. As in the previous chapter, color are described in HSV coordinates than RGB. The reason being that the color in the cylindrical-coordinate representation is more easily clustered than in RGB space [154]. To build the HSV descriptor, histograms were extracted representing the frequency of

Hue, Saturation and Value planes respectively (Figure 5.2 right). As reported by Zarit et al. [155], a system which uses only the Hue and Saturation information is more stable with respect to differences in illumination and local variations caused by shadows. The decision of whether to include the Value component to the color descriptor, in order to improve the detection accuracy, depends on the realism of the environment. If global illumination algorithms (e.g. shadows or radiosity) are implemented, the inclusion of the Value information may degrade the performance of the detection mechanism as there is no easy way to predict such color dynamics by only looking at the camera-object interaction. To detect shadow malfunctions we propose a different approach, reported in Chapter 6. As we shall see later in this chapter, we will show how various geometry and color descriptors perform upon various levels of realism and geometric complexity. To extract the information about the target object from the scene, we segment the image at the object level with a technique similar to that used in the previous chapter; we use *matrix maps* and *matrix tables* in order to identify which pixels belong to which objects. For convenience, the figure depicting this process is shown here again (Figure 5.3).

5.2.2 Object Space Segmentation

In order to be useful for real-case scenarios, both geometric and color descriptors should be common to all objects. Since complex virtual environments are typically composed of thousands of objects, the *one-descriptor-fits-all* solution seems to be very practical. Given that we do not want our descriptors to be any more complex than they already are (we seek to reduce the dimensionality) we try to reduce the complexity of the objects instead in order to appropriately describe complex 2D shapes[1]. In particular, we will split the geometry of the objects into parts that are small enough to be accurately described by our feature vectors. By doing this, any object becomes an ensemble of simple geometric components, each of which will be represented by our color and geometry descriptors. Splitting the original geometry into sub-geometries is advantageous. On the one hand, the second central moments of sub-geometries, on screen, cannot be less accurate than the ones computed on the entire shape, that is, convex shapes are

[1]Different convex geometries may exhibit similar normalized second central moments, and thus, are circumscribed by similar ellipses. Moreover, the occlusion of background objects from the foreground clutter may change the geometry and color descriptors, thus negatively affecting training. This is the case with landscapes, background walls and sky-boxes which are, typically, constantly occluded by the foreground scene. Finally, some attributes of the descriptors may become non-informative for some objects — such as the centroid position of the landscapes or the sky — as they may tend to acquire constant values throughout the game-play experience.

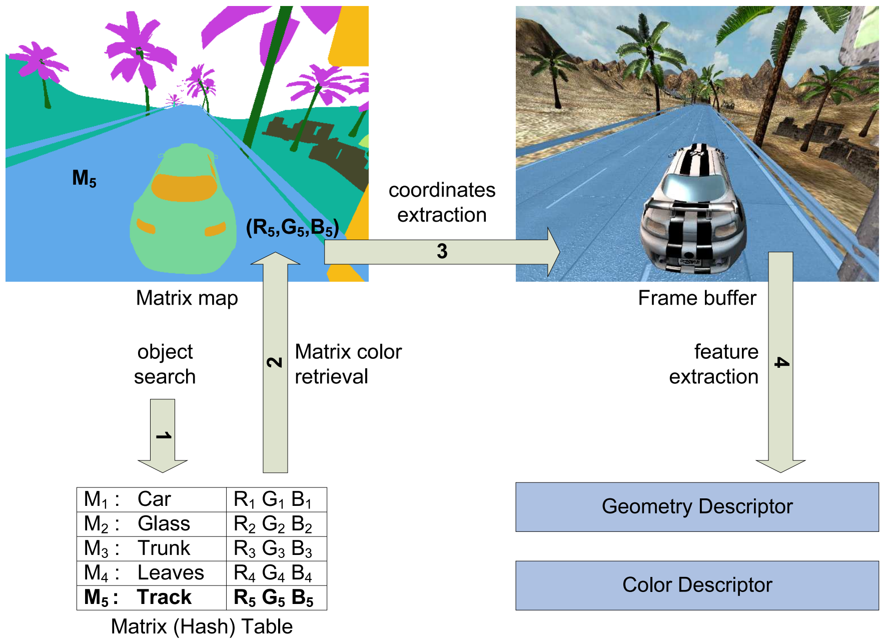


Figure 5.3: Extraction of matrix and pixel information - In order to identify which pixels belong to which objects, matrix maps and matrix tables are used. The pixel coordinates (x, y, z) are taken from the frame and depth buffer. The matrix M_{wvp} is retrieved from the game engine and stored in the matrix table, along with its color. The color of the matrix is then read from the matrix map and used to identify the position on screen of the target object. Finally, color and geometry features are extracted from the frame at the position determined via the matrix map.

likely to be split and form smaller concave silhouettes. Different concave silhouettes are likely to exhibit different second central moments which will produce different circumscribing ellipses. On the other hand, the area of sub-parts is less likely to be severely occluded compared to the entire object which may span a considerable screen region, thus making the description less influenced by the rest of the scene[1]. As we shall see in Section 5.4, such a geometry segmentation, in some cases, improves the accuracy of the detectors. The question that we seek to answer is, logically, how do we split the object geometry in the first place?

In computer vision, the solution to most object segmentation problems needs to be found in screen space, as the image is typically the only information directly available

[1]Object sub-parts that are severely occluded can be easily detected by comparing the observed area (in pixels) to the mean area from the training samples. If an object part is severely occluded, nothing can be inferred about its appearance. However, the analysis can still be carried out for the rest of the ensemble.

from the real world. Unfortunately, the ever changing shape and color distribution peculiar to real scenes makes it hard to discriminate between object parts according to some invariant (shape or color) description of them. However, given that the geometry of virtual objects never changes in local space, segmenting any object simply reduces to segmenting the object space into a number of 3D clusters. Each cluster representing some sub-part of the entire geometry.

We have already seen how this object segmentation can be carried out for the model based classifier. Recall from Chapter 4 (Section 4.4.2, Algorithm 4.4) that we can identify objects parts through a *cluster map* L indicating where the objects parts are on the screen. Here, however, we also want to make this segmentation screen-resolution-independent; because geometry and color features will be extracted in screen space, different screen resolutions may produce different features. One way of doing this is described by Algorithm 5.1.

Algorithm 5.1: Geometry Segmentation

```
1  function SEGMENT($P'_o$,$P_o$,$M_{wvp}$,$r$) returns a matrix of connected components
     inputs:
           matrix $P'_o$ of pixel positions in object space, of size $k \times 4$
           matrix $P_o$ of cluster center coordinates in object space, of size $n \times 4$
           World-View-Projection matrix $M_{wvp}$, of size $4 \times 4$
           resolution $r$, in pixels, of the NDC frame
     locals :
           matrix $N$ of pixel positions in NDC, of size $k \times 4$
           index array pixelClstIdx of clustered pixels, of size $k$
           index array currPixelClstIdx of pixels belonging to the current cluster
           array fRow of the $Y$ pixels component in NDC, of size $k$
           array fCol of the $X$ pixels component in NDC, of size $k$
           matrix $L$ of $n$ connected components in NDC, of size $r \times r$, initially zero
2    $N \leftarrow$ ROUND($(r-1)(P'_o \times M_{wvp} + 1)/2$)
3    for $i \leftarrow 1$ to $k$ do
4        pixelClstIdx[$i$] $\leftarrow$ index of the minimum value of $\|$ROW$(P'_o, i) - P_o\|$
5    end
6    for $c \leftarrow 1$ to $n$ do
7        currPixelClstIdx $\leftarrow$ indices $j$ of pixelClstIdx such that pixelClstIdx[$j$] $= c$
8        fRow $\leftarrow N$ (currPixelClstIdx; 2)
9        fCol $\leftarrow N$ (currPixelClstIdx; 1)
10       $L$ (fRow, fCol) $\leftarrow c$
11   end
12   return $L$
```

As for the model based classifier, here we assume that the position P'_o and the world-view-projection matrix M_{wvp} are available, in object space. We have previously shown how pixels coordinates can be transformed from screen space to object space, through M_{wvp} (Equation 4.2 and 4.3). The cluster centers position matrix P_o contains a number of reference points (rows of the matrix) approximating the consistent geometry of the object, in object space. As for the model based classifier, we shall refer to this matrix as the *database cluster centers* matrix. We compute such a matrix through the Algorithm 4.3 presented in the previous chapter.

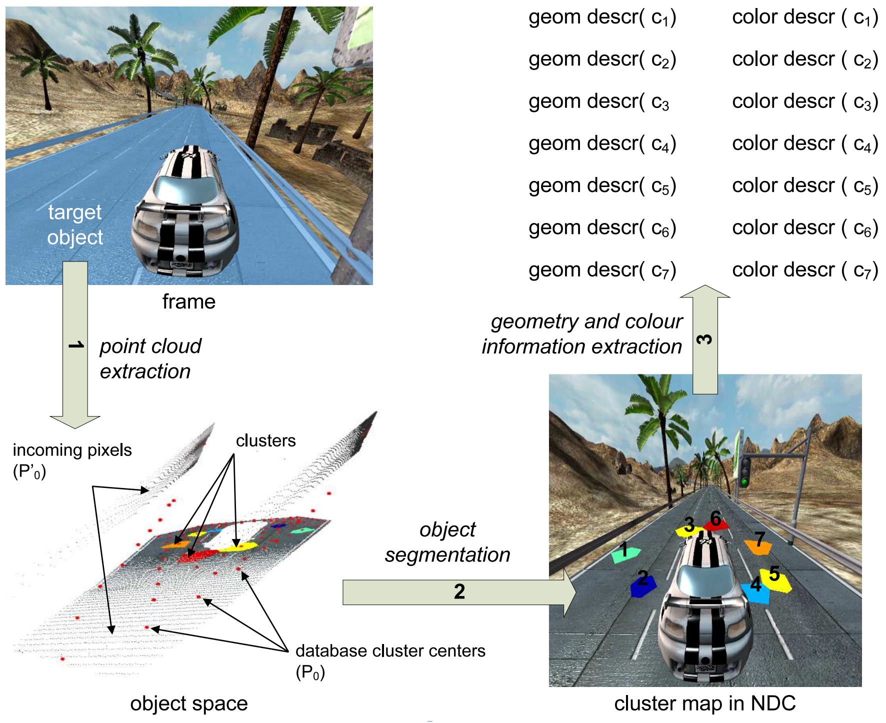


Figure 5.4: Object segmentation - To improve the accuracy of the detection algorithms, an accurate object segmentation is required. To that end, the object (the track in this example) is first split into clusters of pixels, in object space. The cluster centers (bigger red dots) are stored in the database cluster centers matrix P_o. The extraction of features is then performed, in screen space, over the regions of the cluster map L, defined in NDC. In this example, only the first 7 clusters are represented, resulting from a cluster radius of size 15.

The function SEGMENT produces a square matrix, L, of $r \times r$ integers representing the various components of the target object in *Normalized Device Coordinates* (NDC), that is, in coordinates that do not depend on the final screen resolution. Such a *device* is represented by the square matrix N (line 2) computed by first converting the incoming pixels to clip-space (operation $P'_o \times M_{wvp}$). Such a space is defined in the interval $[-1, 1]$ for both x and y dimensions. The division by 2 then shrinks the 2×2 device to size 1×1, after translating its origin from $(-1, -1)$ to $(0, 0)$ (operation $+1$). Finally, the parameter r determines the resolution of the normalized device. To avoid aliasing problems, r should be smaller than both the vertical and horizontal resolution of the screen. The incoming pixels are then clustered and the cluster map generated, as explained in the previous chapter.

Once the segment map is computed, it can be used to segment the object in NDC. From such segments, geometry and color descriptors are extracted as shown in Figure 5.4. Note how that the various object parts are segmented according to their 3D geometry rather than their 2D appearance. This is accomplished by clustering the screen space pixels according to the nearest — in the sense of Euclidean distance — database cluster center in object space. The result of such a clustering is written in the cluster matrix L, which is used to perform a feature extraction on the individual sub-parts rather than on the entire geometry.

5.3 Consistent Appearance Acquisition

In the previous sections, we have seen how objects can be described in screen space. The camera-object interaction will need to be described as well, through vectors or descriptors. This will enable us to define the feature spaces for our estimators and classifiers.

From the previous chapter, we know that the *world-view-projection* matrix M_{wvp} is used by the 3D application for synthesizing the object shape in screen space. Such a matrix is the result of the multiplication of two affine transformations — typically know as *World* and *View* matrices — and one perspective projection. The combined affine World and View transformation defines the translation, rotation and scaling of the object with respect to the camera; the perspective transformation maps the three dimensional vertices of the object to the two-dimensional image we perceive. Therefore, both the original object position in local space and the combined matrix M_{wvp} given by $M_w \times M_v \times M_p$ fully defines the position and shape of the object in screen space. The vectorized version of M_{wvp} is the camera-object descriptor we are looking for. We are now ready to define our feature spaces for visual consistency detection.

5.3.1 Connectionist Models of Consistent Visualization

Partially inspired by the biological processes underpinning perception, connectionist techniques have been developed and effectively employed for pattern recognition tasks such as speech recognition [15], natural language processing [98] and time series prediction [47]. Connectionism refers to those tools, such as Artificial Neural Networks (ANN) and Self-Organizing Maps (SOM), that model mental or behavioural phenomena as the emergent process of interconnected networks of simple units [42]. Such networks have been shown to exhibit interesting aggregate properties. For example, they can be wired to recognize patterns, to exhibit rule-like behavioral regularities and to realize virtually any mapping from patterns of input parameters to patterns of output parameters. Such learning characteristics inspired us to investigate the use of ANNs and SOMs for building the estimators and clustering approaches we introduced in Section 5.1.

5.3.2 Modelling Object Appearance through Feed-Forward ANN

Neural networks are graphs composed of nodes of units connected by directed links. Numerical values (weights) are attached to the links of the graph, parameterizing the input/output function that the network represents. Each unit computes a weighted sum of its inputs and applies an *activation function* to this sum to derive the output. Thus, the output of each unit i is computed as follows:

$$in_i = \sum_{j=0}^{n} W_{j,i} a_j \tag{5.1}$$

$$\begin{aligned} a_i &= g\left(in_i\right) \\ &= g\left(\sum_{j=0}^{n} W_{j,i} a_j\right) \end{aligned} \tag{5.2}$$

where $W_{j,i}$ is the weight associated to the link from unit j to unit i and g is the activation function.

There are two main categories of network structures: *feed-forward* networks and *recurrent* networks. A feed-forward network describes a function of the current input with no internal states; the recurrent network, instead, feeds its output back into its own inputs [123]. Recurrent networks are often used to learn temporal input patterns; the response of the network to a given input depends on its initial state, which may depend on previous inputs.

Neural Networks are typically arranged in *layers*, such that each unit receives input only from units in the immediately preceding layer (left side of Figure 5.6). Learning, in multilayer feed-forward networks, is carried out by *back-propagating* the error $Err = y - h(x)$ from the output layer to the *hidden* layer. Here, y is the *output target* related to the example input x, that is, the output that the network is expected to produce upon x; $h(x)$ is the actual response of the network to x (Figure 5.5). A thorough description of the learning mechanisms for ANNs is beyond the scope of this document; the interested reader is referred to [11]. As has been shown, the standard multilayer

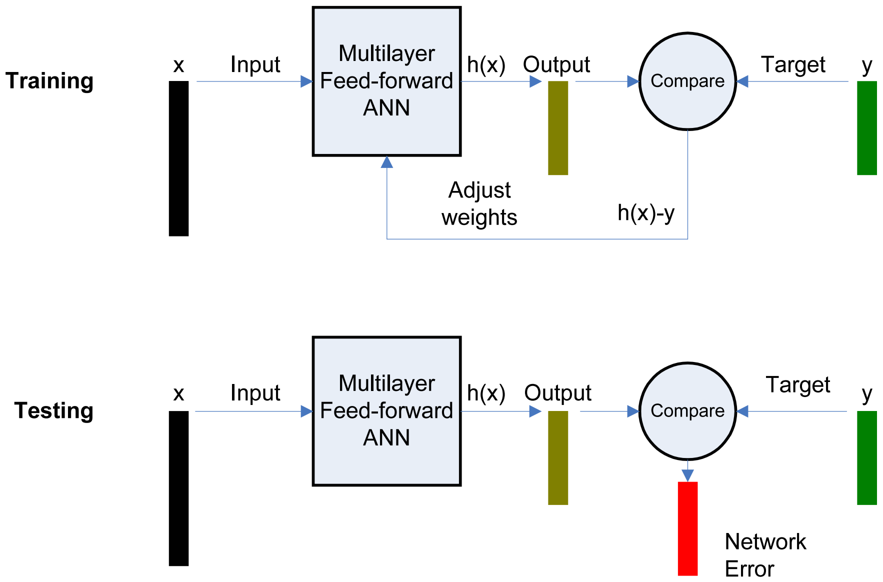


Figure 5.5: Training and Testing in ANNs - Learning in ANNs is carried out via adjusting the network weights according to the error between target and actual output. Once learning has been completed, the network can be used to predict novel data from new stimuli. Illustration inspired by Demuth and Beale [31].

feed-forward network with a single hidden layer can approximate any function (with a finite number of discontinuities) arbitrarily well [28]. As such, this connectionist technique represents a good candidate for solving our regression problem of mapping camera-object interaction (defined through the world-view-projection matrix) to object appearance (defined through geometry and color descriptors).

To train the network, we will feed the input units with the vectorized version of the 4×4 transformation matrix M_{wvp} and correct the network weights according to the target descriptors extracted from the frame rendered through M_{wvp} (Figure 5.6). If the network performs well on the *test* set (i.e. the set of descriptors different from the set used for training the network), we can claim that a good mapping between M_{wvp} and the target matrix exists and it can be expressed through the network. Thus, the trained network can be used for predicting the object appearance and therefore to detect possible visual inconsistencies. Both training and testing stages are depicted in Figure 5.5. The performance of a trained network can be measured by the error values

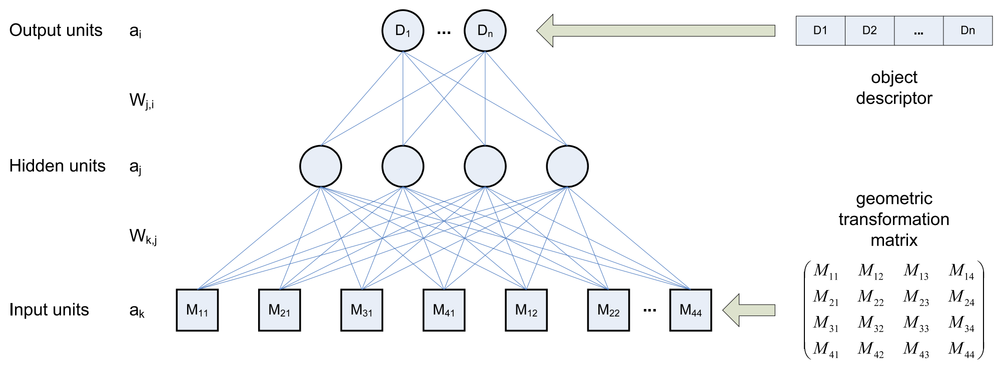

Output units
a_i
$W_{j,i}$
Hidden units
a_j
$W_{k,j}$
Input units
a_k
D_1 ... D_n
M_{11} M_{21} M_{31} M_{41} M_{12} M_{22} ... M_{44}
D1 D2 ... Dn
object descriptor
geometric transformation matrix
$\begin{pmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{pmatrix}$

Figure 5.6: ANN Architecture - In this work, ANNs are trained on the vectorized transformation matrices used to render the object. Outputs are the geometry or color descriptors extracted from the final image.

returned on the training and test sets. However, it is often useful to perform a regression analysis between the network response and the corresponding targets. To this end, the Pearson [109] correlation coefficient, r, is usually computed between network output and corresponding targets. If the outputs are always exactly equal to the targets we have a *perfect fit* and r becomes equal to 1. Likewise, a badly performing network typically exhibits a correlation coefficient close to 0.

5.3.3 Appearance Modelling through Self Organizing Maps

Another type of network which falls into the family of connectionist approaches is the *Self Organizing Map* (SOM). SOMs, also known as Kohonen networks, are unsupervised ANNs involving the non-linear projection of some high-dimensional input space into a low-dimensional discrete output space, typically, a two-dimensional grid. As for an ordinary ANN, self-organizing maps consist of units called nodes or neurons. Associated with each node is a weight vector of the same dimension as the input data and a position in the map space. The goal of training a self-organizing map is to cause different parts of the network to respond similarly to certain input patterns [77]. Therefore, similar samples are mapped together and dissimilar sample are mapped apart. Under a different interpretation, a SOM can be seen as a way of generating discrete approximations of the distribution of the training samples. More neurons will point to high density regions of the input space, leaving fewer where the samples are scarce. To train a SOM, a *competitive learning* approach is used. When a training example is fed to the network, its Euclidean distance to all weight vectors is computed. The neuron with weight vector most similar to the input is called the *Best Matching Unit* (BMU).

The weights of all neurons within a certain neighborhood $N_{ci}\,(t)$ of the BMU are then updated as follows:

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \alpha(t)N_{ci}(t)[\mathbf{x}(t) - \mathbf{w}_i(t)]. \tag{5.3}$$

Here the neighborhood function $N_{ci}(t)$ is a kernel function (typically a Gaussian) with maximum value equal to 1 when $i = c$, where i and c are neuron indices. The width of the kernel decreases with time. The scalar α is a monotonically decreasing learning coefficient and $\mathbf{x}$ is the input vector. When a vector $\mathbf{x}$ is presented, the weights of the winning neuron and its close neighbors move towards $\mathbf{x}$ according to N_{ci}. This process is repeated for each input vector for a (usually large) number of iterations. The ability of SOMs to visualize multidimensional data is exploited in many application areas such as process analysis [2], pattern recognition [72] and novelty detection [153]. Figure 5.7 depicts a typical example of SOM novelty detection; such a scheme is the one we use in this work. Specifically, we train SOMs to represent the distribution of the input space in order to make statistical inferences about the correctness of new observations.

Self Organizing Maps and Artificial Neural Networks are inherently different approaches. SOMs are suitable for carrying out clustering, ANNs for performing regression. As we stated earlier, SOMs accept target inputs but no target outputs, that is, they learn without supervisory signals. Ultimately, what SOMs learn is the representation of the input data; in our case, the input and output vectors sampled from the consistent behaviour of the rendering system. To be precise, the vectors we used to train SOMs resulted from the concatenation of the vectorized geometric transformation matrix M_{wvp} and the image descriptors (Figure 5.8). Once the output neurons have moved so as to homogeneously cover this hybrid feature space, the network can be used for classification purposes. As new vectors are submitted to the trained SOM, the Euclidean distance between such vectors and their BMUs is computed. If this distance exceeds a threshold, then the input vector can be labeled as novel and the related object identified as suspicious. Conversely, any vector close enough to its BMU can be considered consistent. As we have shown in Chapter 3, this is an example of SOMs application to novelty detection problems. Note that concatenating M_{wvp} to the geometry or color vectors was a choice motivated by the availability of extra data from the game. SOMs could have been trained and tested on feature spaces defined by color or geometric descriptors only. We did this because the extra information about the geometric transformation endows the vectors with more descriptive power.

The reasons for choosing SOMs compared to other partitional clustering algorithm

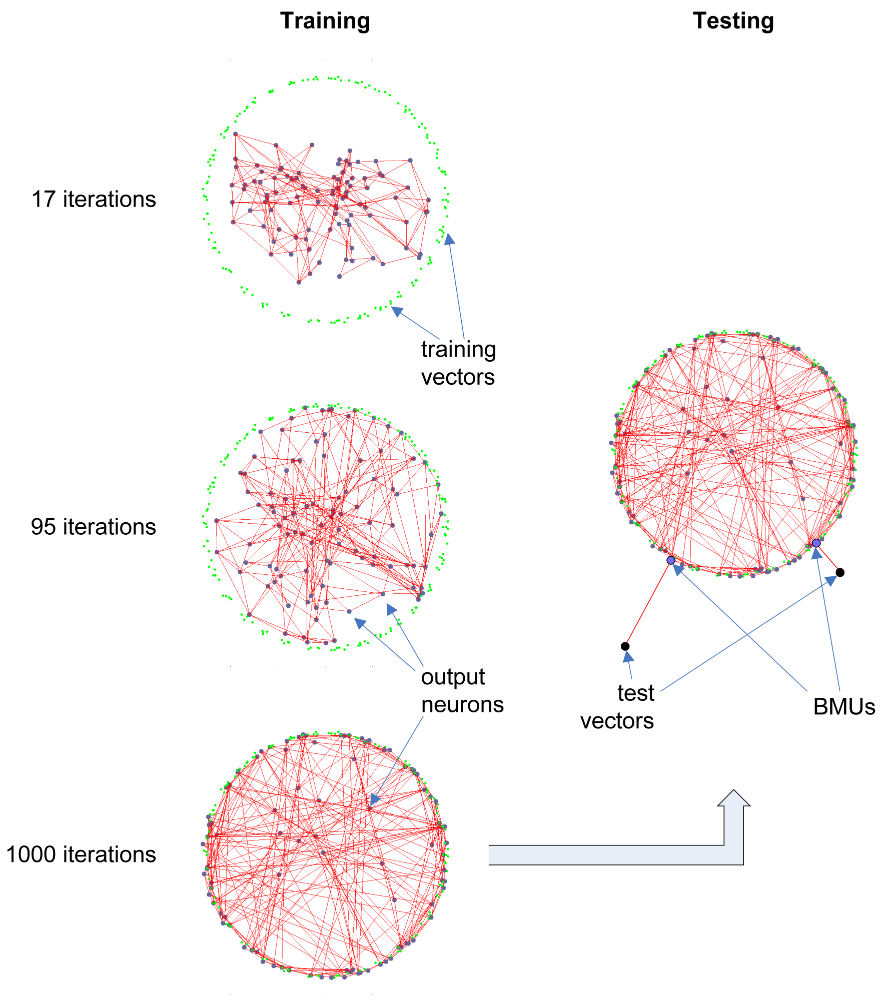


Figure 5.7: Training in SOMs - Neurons in SOMs move from their original position so as to cover the input space. Once a high enough number of iterations has been reached, the network can be used for novelty detection purposes. This can be achieved by using the distance (weight) between a test vector and its BMU. If such a distance exceeds a threshold, then the new vector can be considered suspicious. This figure depicts an example of SOM training performed in our datasets. Dots in light green represent feature vectors of the color feature space; dots in dark blue represent neurons of the SOM. To visualize the feature vectors, these were projected to the first two major components of the (10 to 20 dimensional) feature space, extracted through Principal Component Analysis (PCA). Black dots, indicating anomalous vector, were manually added in the picture for clarity purposes.

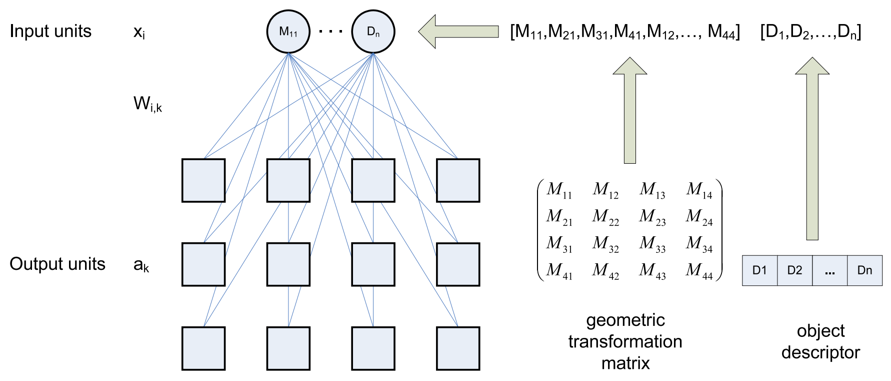

Input units
x_i
$W_{i,k}$
Output units
a_k
M_{11}
D_n
$[M_{11},M_{21},M_{31},M_{41},M_{12},\ldots, M_{44}]$
$[D_1,D_2,\ldots,D_n]$
$\begin{pmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{pmatrix}$
D1
D2
...
Dn
geometric transformation matrix
object descriptor

Figure 5.8: SOM Architecture - The SOMs are trained on the combined vectors resulting from the concatenation of the vectorized transformation matrix and the geometry or color descriptor. During training, the output neurons will move from their original position in order to best represent the input training set.

(e.g *K-means*) are manifold. First, due to the effects of the neighborhood function, SOMs are likely to be less prone to local optima than K-means [6] or, to put it another way, SOMs are quite insensitive to initial conditions [49]. Second, learning in SOMs can be performed in an on-line mode, that is, by adding new consistent vectors as soon as they become available [107]. Finally, the topological map can be used for visualization purposes; the clustering resulting from SOMs is more visual and easy to comprehend and analyze.

5.4 Implementation and Results

In the previous sections we have proposed two different classifiers to assess the consistent behavior of the rendering system. One based on the modelling of some behaviour of the rendering system through ANNs; the other, consisted in clustering the hybrid feature space of world-view-projection matrices and object descriptors, through SOMs. Both approaches have been implemented and thoroughly validated on three different versions of the Microsoft Racing Game environment presented in Chapter 2. The results are presented in the following sections.

5.4.1 Estimator Parameters

The ANNs we used in our experiments were two layer neural networks with a variable number of hidden units. Preliminary tests showed that good predictions could be obtained from networks with a tan-sigmoid transfer function in the hidden layer and linear function in the output layer. The training algorithm we employed is called *Bayesian Regularization* which statistically estimates the regularization parameters of the network. A detailed discussion of such a training technique can be found here [92]. The choice of one training function over another generally depends on the data set. The Bayesian Regularization turned out to be a good choice, in terms of accuracy, for the environments and objects we considered in our work.

The entire data set for ANNs, composed of transformation matrices M_{wvp} (input) and geometry and color vectors (output), was split into a training set (80 percent of the data set) and a validation set (20 percent of the data set). The maximum number of passes through the entire training set (epochs) was set to 1000, whereas the maximum number of consecutive *validation failures* was set to 6[1]. Maximum number of epochs and validation failures were chosen in such a way as to ensure that the ANNs always converged to a solution (the network underwent *early stopping*) before the maximum number of epochs was reached. The performance on the validation set was computed via *Sum Squared Errors* (SSE).

As far as concerns SOMs, we used a single layer of a variable number of output units. As learning in SOMs is unsupervised, there was no need to divide the data set into training and validation sets, thus, the whole data was used for training. As stated earlier, the training vectors for SOMs result from the concatenation of vectorized transformation matrices M_{wvp} and geometry or color descriptors.

In both SOMs and ANNs, weight and bias updates occurs at the end of an entire pass through the input data (batch update). Table 5.1 shows the parameters that were set to a fixed value throughout our tests.

Parameter	ANN	SOM
max epochs	1000	1000
steps to shrink to 1	N/A	100
initial neighborhood size	N/A	3
error function	SSE	N/A
max validation failures	6	N/A
topology	2 layers	1 layer (hexagonal grid)

Table 5.1: Parameter set for ANNs and SOMs

[1]In ANN terminology, validation failures are increases on the validation error, compared to the previous epochs. The maximum number of failures is used to control overfitting. When the validation error increases for a specified number of iterations, the training should be stopped as that is an indicator that the network is learning noise in the data [31].

5.4.2 Dimensionality Reduction for the Geometric Transformation Matrix

The World-View-Projection matrix we use to feed our networks has full rank, that is, its columns are linearly independent. However, the linear independence between columns does not guarantee the non-redundancy of the related dimensions, taken singularly, across the entire data set. As it turns out, not all elements of the matrix are equally significant. To see this numerically, we performed a Principal Component Analysis (PCA) on the entire data set of transformation matrices. We found that, insofar as the collection of validated frames is concerned, as few as 6 out of 16 principal components could explain almost the entire variation (more than 95%) of the data set and that the first 12 principal components were enough to explain the entire variation (100%). This can be observed from the *Pareto* chart depicted in Figure 5.9 (a), in which the bars indicate the variance explained by each component and the line graph denotes the cumulative sum over the principal components up to 95% of the entire variation. This left us with the decision of using the first 12 values of the vectorized World-View-Projection matrix for both training and test.

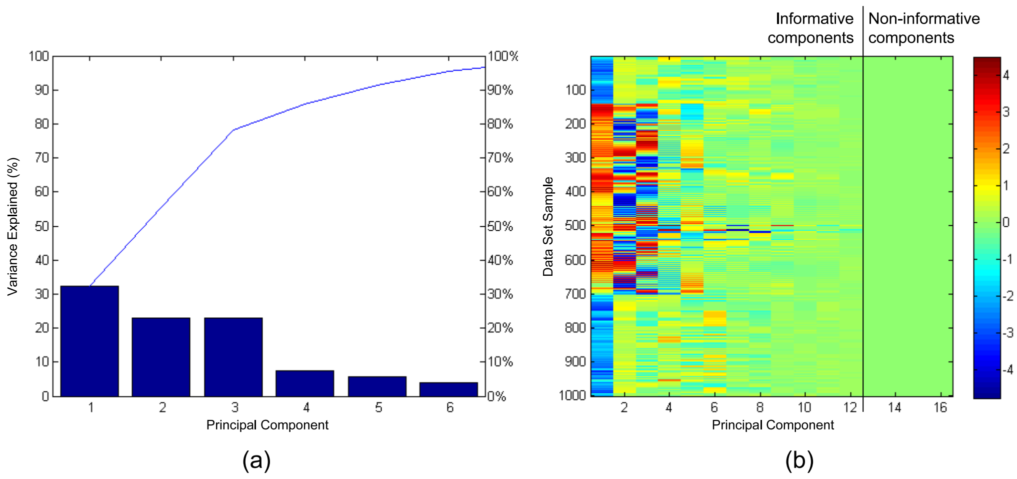


Figure 5.9: PCA of transformation matrices - World-View-Projection matrices, in computer graphics, have full rank. However, the data emerging from a number of play sessions, over different objects, shows that 6 out of 16 dimensions suffice to explain most of the variance of the entire data set of geometric transformations (a). When such vectors are transformed into the new coordinate system of principal components, it becomes visible how dimensions 13 to 16 (the last column of the transformation matrices) do not contribute much to the variance of the data set (b).

5.4.3 Accuracy Measurements

Recall from Section 5.2.2 that our virtual objects undergo a segmentation in order to reduce the visual complexity of the entire entity. Thus, what we assess is not the object as a whole, but its individual components, as both the geometry and color consistency is, in effect, an independent property of the individual object parts. The aim of this section is to show the difference, in performance, between ANNs and SOMs. Furthermore, we shall show that our experiments support the hypothesis that splitting the original object geometry in sub-parts does, in some cases, increase the overall accuracy of the system.

For each component, we trained four networks: two (ANN and SOM) for learning the correct geometric appearance and two for capturing the correctness of color distribution. The effectiveness of our classifiers is measured through ROC (Receiver Operating Characteristic) curves, explained in the previous chapter. In particular, the Area Under Curve (AUC) was used as a measure of accuracy. As for the model based classifiers, to compute true positive rate (TPR) and false positive rate (FPR), *true* sets and *false* sets were generated, containing bug-free and buggy visualizations respectively. To facilitate the comparison between the model based and the view based approach, training and testing for the view based classifiers was performed on the same dataset used for the model based classifiers. Bugs in the false set were introduced as explained in the previous chapter (see Figure 4.14). Figure 5.10 shows examples of ROC curves used to measure the accuracy of the view based classifiers.

Tests have been carried out on three different geometries and levels of realism. These are the ones used for validating the model based classifiers, shown here again for convenience (Figure 4.13).

5.4.3.1 Geometry Tests

To validate the capability of our system to detect geometry bugs, we performed an extensive analysis on a number of different network configurations, geometry descriptors and cluster sizes. We effectively evaluated 300 different scenarios by combining:

- Five different ANN and SOM architectures
- Fifteen different geometry descriptors
- Four different cluster radii

The networks we analyzed were 8, 12, 16, 20 and 24 hidden unit ANNs and 6×6, 8×8, 10×10, 12×12 and 14×14 output grid SOMs. The shape of the objects on

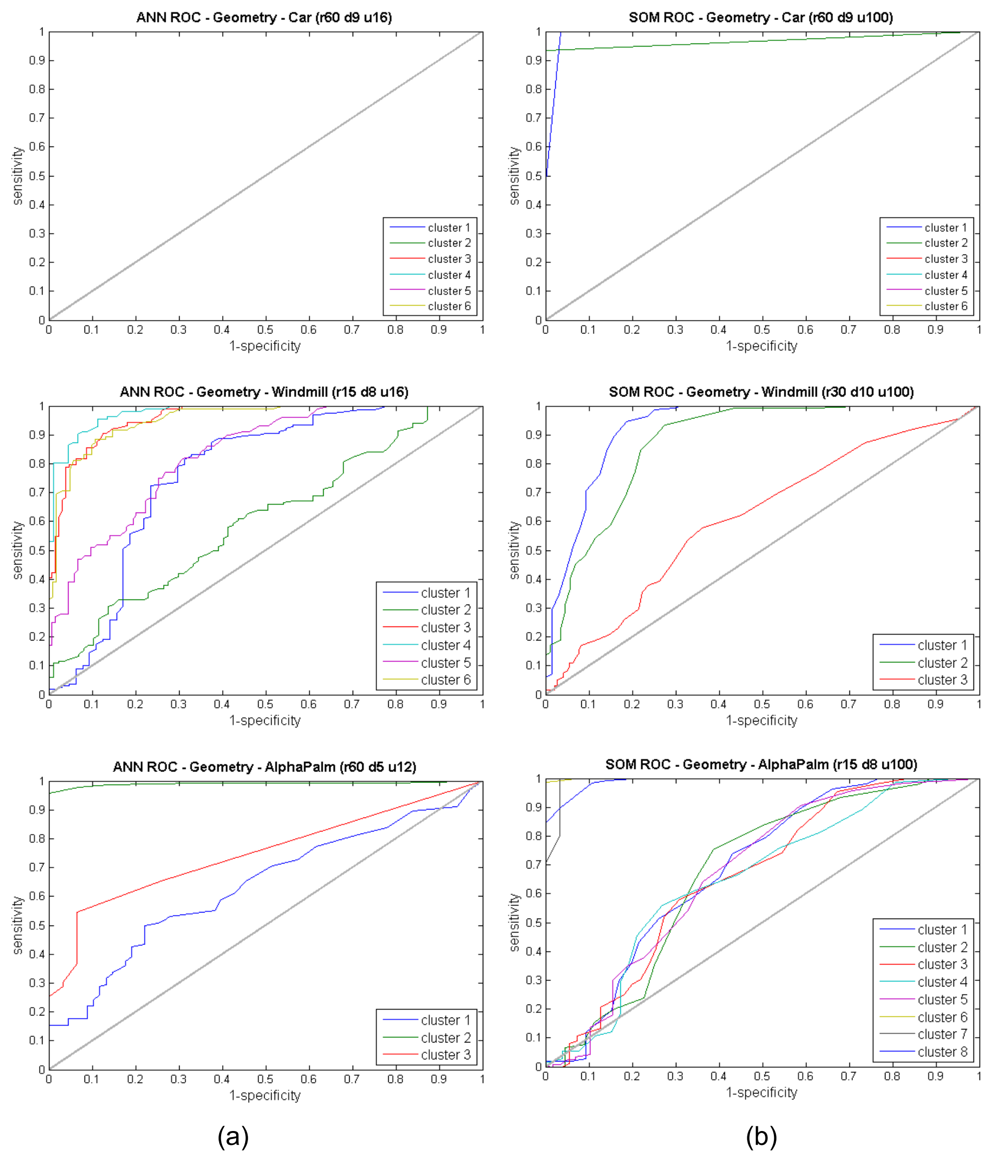


Figure 5.10: ROC curves from the view based classifiers - The graphs show the ROC curves obtained from geometry test performed on three different objects (Car, Windmill and AlphaPalm). Column (a) relates to ANN tests, column (b) to SOM tests. Different curves within the same graph represent different clusters (sub parts) in which the original geometry was segmented. Because sub parts may have different geometrical complexity and size, performance may vary amongst them. Curves lying along the diagonal straight line indicate a poorly performing system. By contrast, curves closer to the upper-left corner of the ROC plot indicate highly accurate systems. The graphs title contain information about the radius, descriptor and network architecture used. For example, (r60 d9 u16) indicates that the system has been trained and tested with radius 60, descriptor 9 and via using a neural network of 16 hidden neurons (output neurons in case of SOMs). Note, from the first row, how convex geometries (i.e. the car) are likely to be learned by the system and be effectively differentiated from anomalies. Conversely, the appearance of more complex, concave geometries (i.e. the palm tree) is less likely to be accurately learned by either ANNs or SOMs.

screen was defined by the 15 descriptors depicted in Figure 5.11. Finally, in order to test our hypothesis of reducing the clusters size for enhancing accuracy (Section 5.2.2), we allowed for 5 different cluster radii (see Algorithm 4.3) of size 15, 30, 60 and 120.

Num.	Area	C.x	C.y	Depth	E.max	E.min	E.theta
1	Yes	No	No	No	No	No	No
2	No	Yes	Yes	No	No	No	No
3	Yes	Yes	Yes	No	No	No	No
4	No	No	No	Yes	No	No	No
5	Yes	No	No	Yes	No	No	No
6	No	Yes	Yes	Yes	No	No	No
7	Yes	Yes	Yes	Yes	No	No	No
8	No	No	No	No	Yes	Yes	Yes
9	Yes	No	No	No	Yes	Yes	Yes
10	No	Yes	Yes	No	Yes	Yes	Yes
11	Yes	Yes	Yes	No	Yes	Yes	Yes
12	No	No	No	Yes	Yes	Yes	Yes
13	Yes	No	No	Yes	Yes	Yes	Yes
14	No	Yes	Yes	Yes	Yes	Yes	Yes
15	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Geometry descriptors

Num.	Hue (bins)	Sat. (bins)	Value (bins)
1	10	0	0
2	12	0	0
3	14	0	0
4	16	0	0
5	18	0	0
6	20	0	0
7	8	2	0
8	9	3	0
9	10	4	0
10	11	5	0
11	12	6	0
12	13	7	0
13	6	2	2
14	7	3	3
15	8	4	4
16	9	5	5
17	10	5	5

Color descriptors

Figure 5.11: Descriptor tables - In order to validate the our system we analyzed the behaviour of various network architectures with several geometry and color descriptors. In the geometry descriptor table, the values *yes* and *no* indicates whether the descriptor includes the related component or not. The values within the color descriptor table denote the number of bins used to build the histogram of the related component.

As with the model based classifier, results are displayed through box plots and decision trees. Box plots and decision trees help to visualize and interpret our test data, as explained in the previous chapter. Note from Figure 5.12 and 5.13 that both ANNs and SOMs can achieve high accuracy, upon selecting the appropriate parameter set. In particular, note that cluster radius and geometry descriptor play a role in the accuracy of the system. As far as concerns ANNs, descriptors 3, 7, 8, 10, 11, 12, 14 and 15 seem to perform better than the others (see top nodes of the decision trees). This indicates that the area of the object is a good feature to include, but only if combined with object centroid and depth (descriptor 3 and 7). Likewise, the eccentricity information results being a good feature, either alone (descriptor 8) or combined with other features (as for descriptors 11, 12 and 14). By contrast, SOMs are less sensitive to the descriptor used and perform slightly worse only if descriptor 4 (the one containing depth information only) is employed.

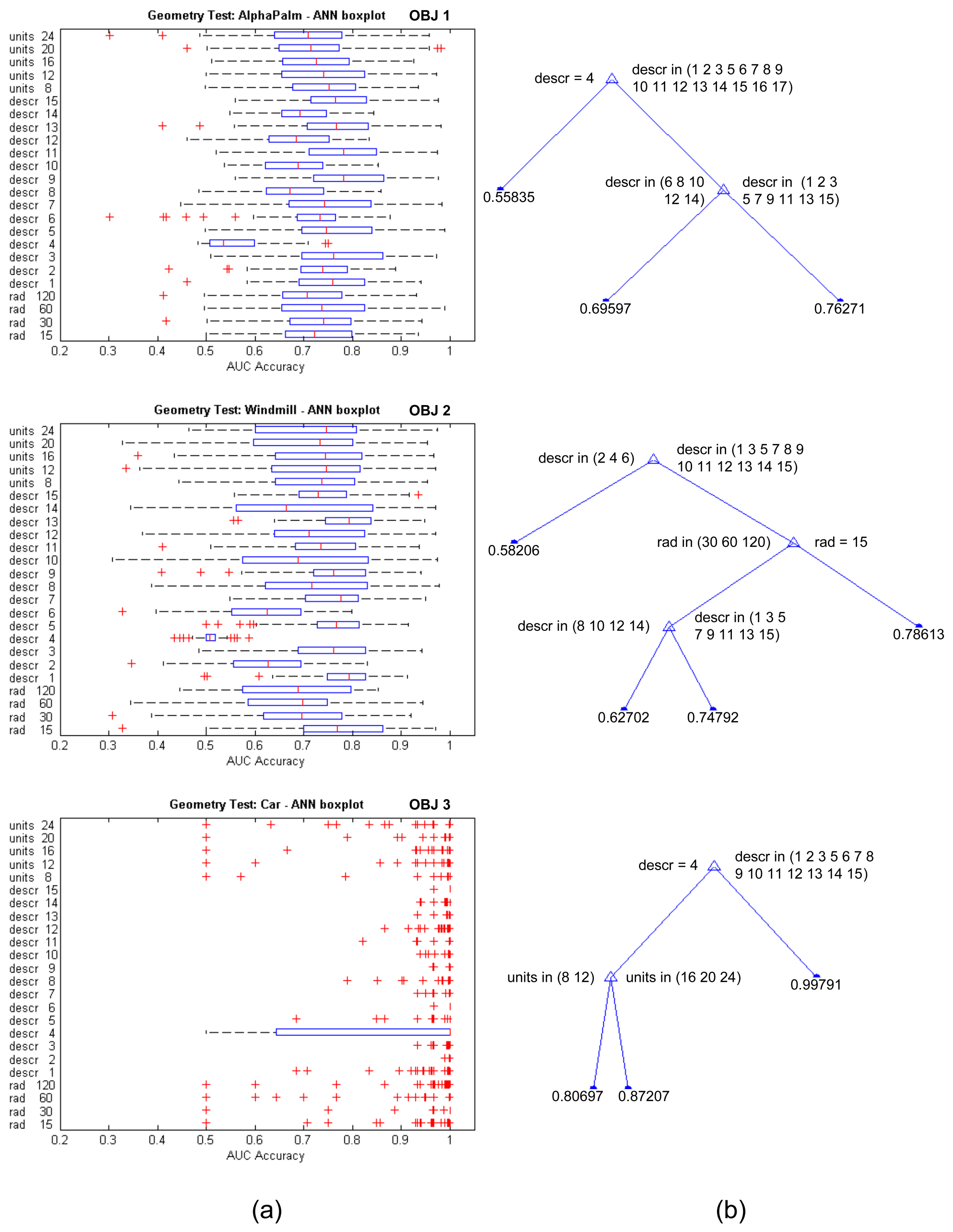
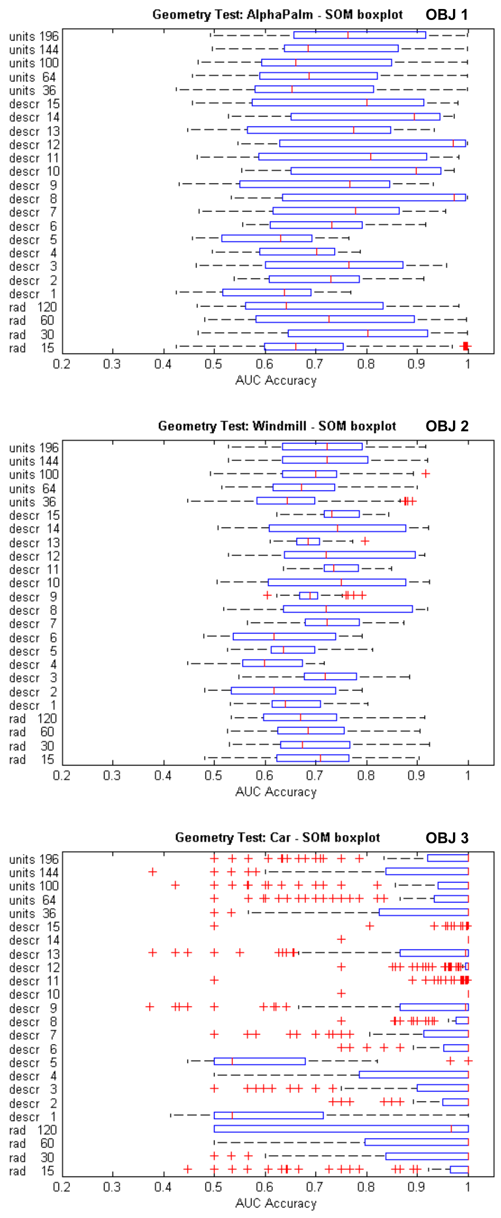
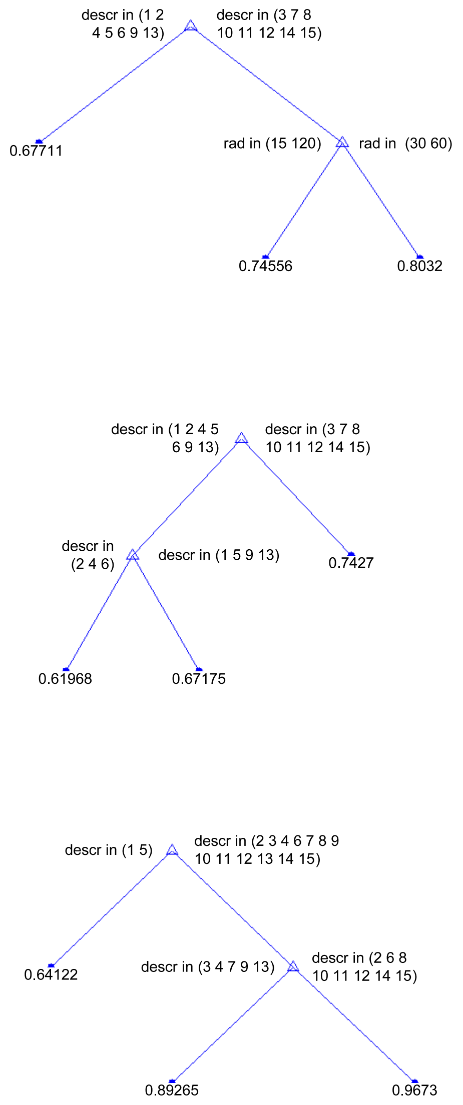


Figure 5.12: ANN geometry tests - Box plots and decision trees were used to interpret the results from the validation of our classifiers. From the box plots it is evident that geometry descriptor (parameter *descr*) significantly influences the accuracy of the system. In particular, descriptor 4 is likely to degrade the system accuracy. The cluster radius (parameter *rad*) is also an important factor, as far as accuracy is concerned. Indeed, the decision tree related to the object windmill (second row), uses the radius as a decision node. Finally, the network architecture (parameter *units*) seems not to play a major role in the system accuracy. Although the median of the box plots changes with respect to the variable *units*, the difference between sample means is not statistically significant. See text for more details.

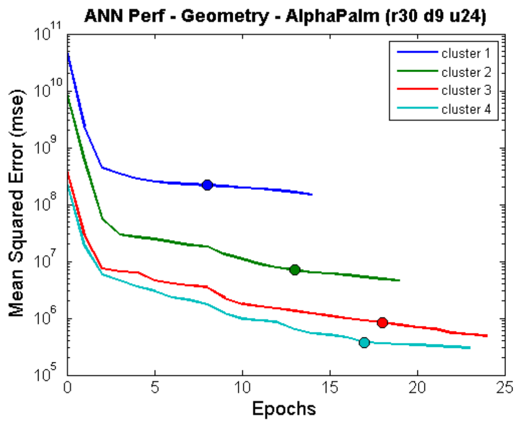


(a)

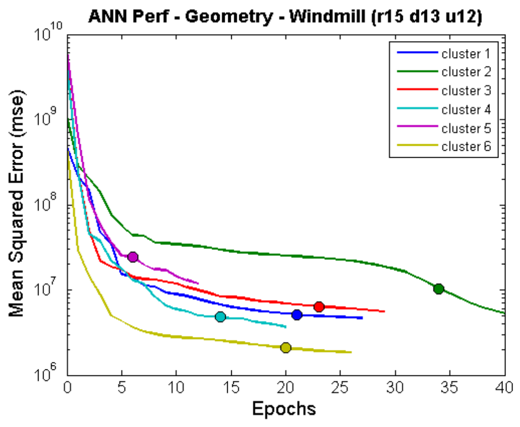

descr in (1 2 4 5 6 9 13)
descr in (3 7 8 10 11 12 14 15)
0.67711
rad in (15 120)
rad in (30 60)
0.74556
0.8032
descr in (1 2 4 5 6 9 13)
descr in (3 7 8 10 11 12 14 15)
descr in (2 4 6)
descr in (1 5 9 13)
0.7427
0.61968
0.67175
descr in (1 5)
descr in (2 3 4 6 7 8 9 10 11 12 13 14 15)
0.64122
descr in (3 4 7 9 13)
descr in (2 6 8 10 11 12 14 15)
0.89265
0.9673

(b)

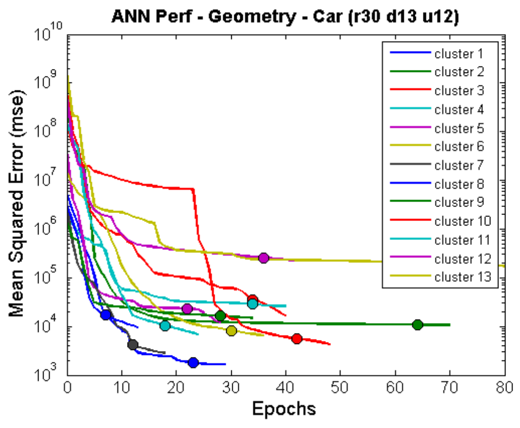
Figure 5.13: SOM geometry tests - The box plots and decision trees for the SOM geometry tests show that the geometry descriptor is an important factor, as far as accuracy is concerned. This result is confirmed by the decision trees, which use the descriptor as a major discriminative factor (root node). The cluster radius is also a discriminative parameter, although this can only be observed from the box plots.



(a)



(b)



(c)

Figure 5.14: Performance of ANNs in geometry classifiers - Trends of the Mean Squared (geometry) Error for the best performing ANNs computed on the training set. Different plots within the same graph represent the error functions for each cluster (and therefore network) of the target object. The dots indicate the time (epoch) at which training is stopped in order to avoid overfitting. Early stopping is achieved via monitoring the network error on the validation set (not represented in the graphs). Accordingly, the training is stopped when the validation error increases for a specified number of iterations (6 in our experiments). The graphs refer to ANNs trained for the objects "AlphaPalm" with radius 30, geometry descriptor 9 and 24 hidden neurons (a); "Windmill" with radius 15, descriptor 13 and 12 hidden neurons (b); and "Car" with radius 20 descriptor 13 and 12 hidden neurons (c).

As for the previous chapter, in order to measure the statistical significance of box plot and decision trees we performed an ANOVA test. Through the ANOVA, we tested whether the mean accuracy produced by a factor (e.g. a specific cluster radius, descriptor or network architecture) is significantly different from the mean accuracy produced by any other factor. A small p value (e.g. 0.01) of the ANOVA test, indicates that the influence of the related factor to the system accuracy is statistically significant. The p values of the ANOVA test are reported in the gray columns.

From the second row of Table 5.4, we can observe that the influence of geometry descriptors to the accuracy of ANNs is statistically significant 5.4. The cluster radius is also discriminative (see first row of Table 5.4), but only for concave geometries (i.e. the palm tree and the windmill). Finally, note that the system accuracy for ANNs, except for the windmill (see forth row of Table 5.4), is not decided by either the network architecture or related to multiple factors (grouping variables). In other words, accuracy is not significantly influenced by the number of hidden units of the ANNs; by the radius and descriptor, taken jointly; or by the radius and network architecture, take jointly; or on the radius, the descriptor and the architecture; and so on and so forth.

Similar results can be observed for SOMs. In particular, note how geometry descriptors and cluster radii greatly affect the accuracy of the system (first and second row of Table 5.5). The p values of the ANOVA test are zero to four decimal places. A statistic as extreme as that observed would occur by chance only once in more than 10,000 times if different factors would produce equal means. As for ANNs, the SOM network architecture (i.e. number of output units) seems not to play an important role, over all (see third row of Table 5.5). Differently from ANNs, however, cluster radius and geometry descriptor can together explain the variance of the accuracy, for the windmill and the car (see forth row of the table).

From our geometry tests we cannot conclude which approach — either ANN or SOM — performs better than the other. Upon selecting an appropriate parameter set, both approaches give good results. The time trajectory of the Mean Squared Error (MSE) for the best performing ANNs is shown by the graphs in Figure 5.14. The graphs also indicate the time (epoch) at which training is stopped in order to prevent overfitting. Table 5.2 reports the parameter sets of the best performing systems for both ANNs and SOMs. The values reported in the table have been automatically determined by considering the best performing branch of the ANN and SOM decision trees. The best accuracy value for each object or rendering level is the one corresponding to the leaf of the rightmost branch of the related non-pruned tree (see Figure 5.12 and 5.13 (b) for a reference). For instance, the best ANN accuracy leaf for the object *Alphapalm* (87.37%) is the one having the nodes {Radius $\in \{30, 60\}$}, {Descriptor $= 9$} and {Units $= 24$} as ancestors. Multiple values for the same parameter (column) indicate that values close to the related accuracy can be obtained through different parameter sets. For example, if the object *car* is to be assessed, a highly accurate geometry inconsistency detector can be built by using either clusters of size 30 or 60; and descriptor 2, 6, 9, 13 or 15; with any network architecture (see third row of table 5.3). As discussed in the previous chapter, when muliple radius choices are available (see first and third row of the table) the biggest values (i.e. 60) should always be selected for building the classifier. Bigger cluster radii produce smaller number of clusters or sub-geometries. A smaller number of clusters entails a smaller number of elements to process and store, thus less computational and memory loads.

Table 5.2: Best performing View Based geometry classifiers

	ANN				SOM			
Object	Radius	Descriptor	Units	Accuracy (AUC)	Radius	Descriptor	Units	Accuracy (AUC)
AlphaPalm	30,60	9	24	0.87375	30	8	144,196	0.90426
Windmill	15	13	12	0.86357	15	7	144	0.80928
Car	30,60	2,6,9,13,15	ANY	1	15	8	100	0.99169

5.4.3.2 Color Tests

The capability of our system to detect color related bugs was measured across 340 different architectures by combining:

- Five different ANN and SOM architectures
- Seventeen different geometry descriptors
- Four different cluster radii

The network architectures, as well as the cluster radii used were the same as the ones used for the geometry tests. The 17 different color descriptors were obtained by combining various Hue, Saturation and Value histograms as shown in Figure 5.11. As for the model based classifier, the view based classifier was validated on one complex object (a palm tree) across three different levels of realism (Figure 4.13). These were:

- Level of Realism 1 (LOR1): diffuse, ambient and specular light contributions
- Level of Realism 2 (LOR2): sharp threshold on diffuse, ambient and specular light components

- Level of Realism 3 (LOR3): diffuse, ambient, specular light contributions; motion blur and bloom

Figure 5.15 and 5.16 show the results of the ANN and SOM classifiers respectively. As it can be observed, there is a noticeable gap between the overall performance of ANNs versus the performance of SOMs. ANNs seem to perform badly, regardless of the parameter set used. The accuracy of the best performing ANNs ranges between 60% and 65%. The decision trees of Figure 5.15 (b) and the ANOVA tests of Table 5.6, do not show any peculiar pattern. Rather, the best parameter set seems to be highly dependent on the target object and type of rendering (level of realism) to test.

By contrast, results from figure 5.16 show that accurate color inconsistency detectors can be built via using SOMs. The cluster radius, in such case, seems to play the most important role. This can be observed by looking at both the decision trees and the ANOVA test in Table 5.7. All SOM-related decision trees have the cluster radius as principal discriminator in the root node. Also, the decision tree show that radii 30 and 60 usually perform worse than the others. The discriminative power of the cluster radius for SOMs is confirmed by the ANOVA tests. The p value is very small (zero to four decimal places) for all type of rendering considered, that is LOR1, LOR2 and LOR3 (see first row of Table 5.7). The color descriptor for SOMs becomes important only in one case, for the level of realism 2 (LOR2) (see second row of the table). In this case, descriptors 1, 2, 3, 4, 5, 6 and 8 give better performances with respect to the other descriptors (see the second row of Figure 5.16 (b)). LOR2 represents the environment rendered through the cartoon-like technique (see Figure 4.13). The aforementioned descriptors are the ones having the Hue component only (except for descriptor 8), as can be observed from Figure 5.11. These results suggest that when non photo-realistic environments are to be assessed, colors can be effectively represented through SOMs and Hue histograms. As the environment becomes more photo-realistic, other color components may need to be included in the descriptors in order to capture color changes, for example, due to specular and reflection effects. The time trajectory of the Mean Squared Error (MSE) for the best performing ANNs is shown by the graphs in Figure 5.17. Table 5.3 reports the best performing ANN and SOM color inconsistency detectors. Note how SOMs are more suitable to describe colors than ANNs. Indeed, while the best performing ANNs can only reach 65% of accuracy, SOMs can be 95% accurate. Also, note how more complex color descriptors are needed to model color in realistic environments, such as LOR1 and LOR3 (see first and third row of the SOM table).

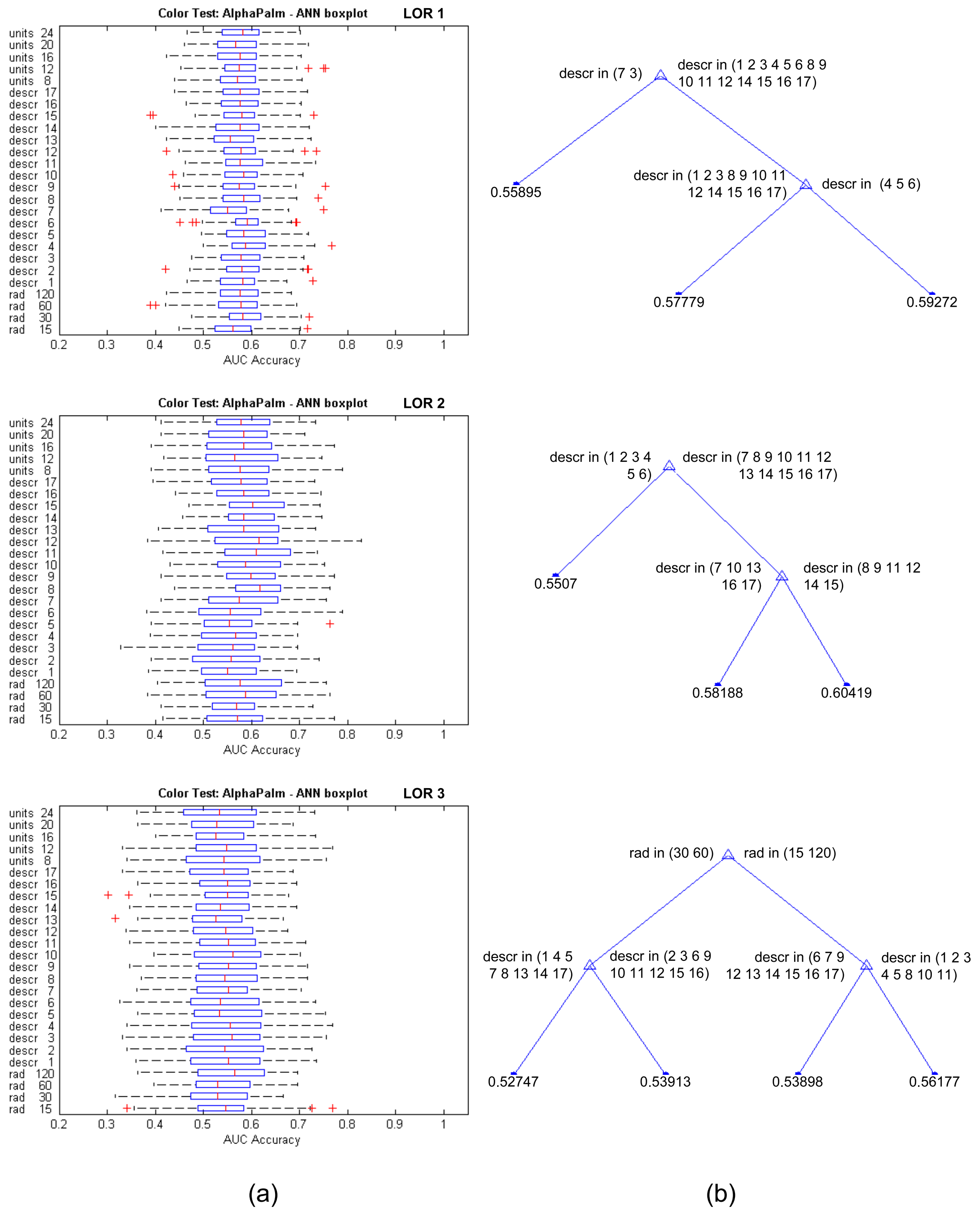


Figure 5.15: ANN color tests - Box plots and decision trees show that ANNs perform poorly in color inconsistency detection. The overall performance of the system is around 60%. Both cluster radius and geometry descriptor are descriminative parameters but there seems not to be a generally good or bad descriptor or radius. The best parameter set depends also on the target geometry.

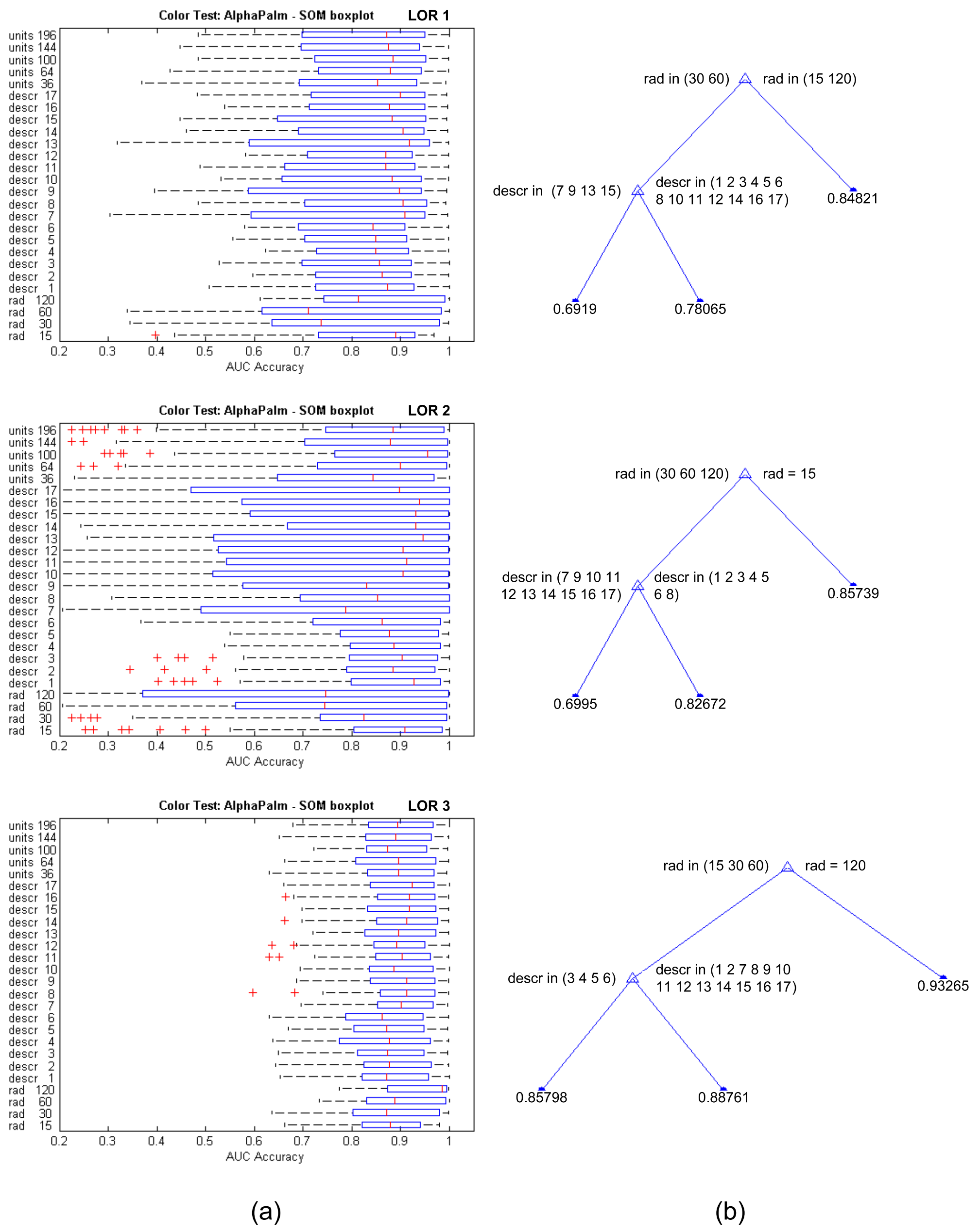


Figure 5.16: SOM color tests - From the box plot and decision tree results of the SOM classifier, it can be noticed that the overall system accuracy is high (about 90%). Both cluster radius and color descriptor are important factors. However, only the cluster radius makes a statistically significant difference overall. See text for more details. This result is confirmed by the decision trees, which consider the cluster radius as the most important predictor (root node).

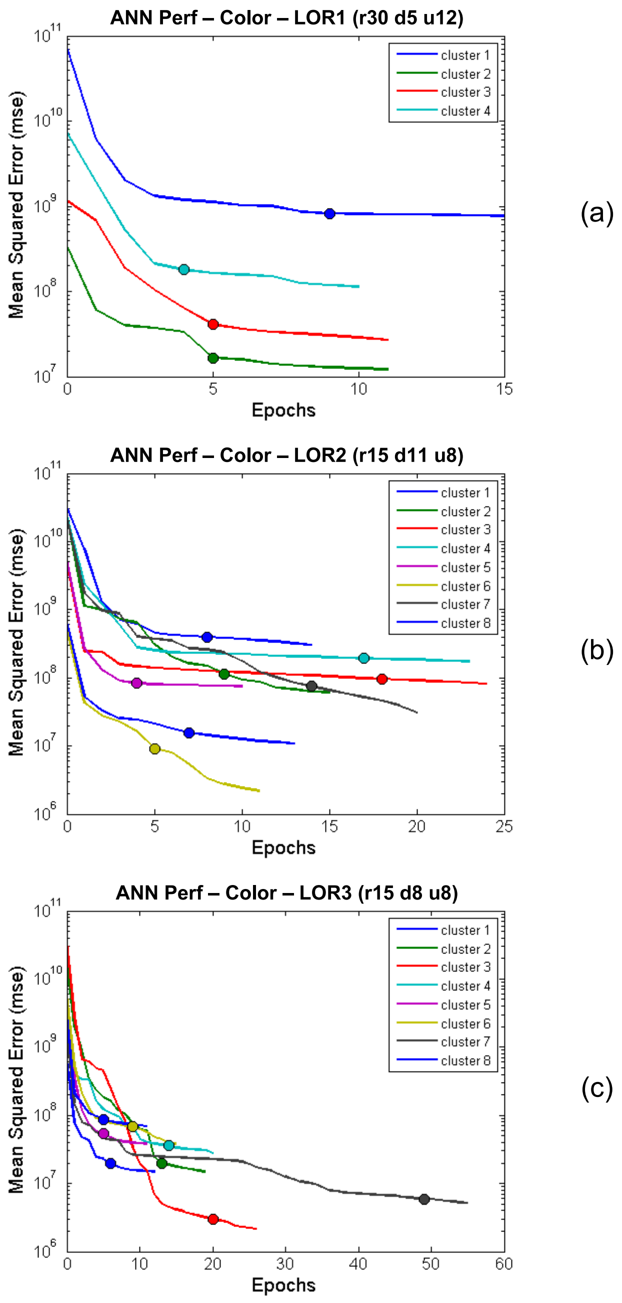


Figure 5.17: Performance of ANN in color classifiers - Trends of the Mean Squared (color) Error for the best performing ANNs computed on the training set. The graphs refer to ANNs trained for the environment "LOR1" with radius 30, color descriptor 5 and 12 hidden neurons (a); "LOR2" with radius 15, descriptor 11 and 8 hidden neurons (b); and "LOR2" with radius 15 descriptor 8 and 8 hidden neurons (c).

Table 5.3: Best performing View Based color classifiers

	ANN				SOM			
Environment	Radius	Descriptor	Units	Accuracy (AUC)	Radius	Descriptor	Units	Accuracy (AUC)
LOR 1	30,120	5	12	0.63602	15	14	100	0.88622
LOR 2	15	11	8	0.65736	15	1	144	0.91342
LOR 3	15	8	8	0.60556	120	8,11,16	196	0.95684

5.4.4 Model based vs. View Based

We began this chapter by asserting that a view based approach to consistency detection was needed in order to allow for color changes due to the camera-object interactions. Recall from the previous chapter, that the model based approach was postulated to be unable to learn significant color changes over the object surface. The purpose of this section is to put this hypothesis to test.

We performed tests on the environment LOR1 which featured high quality rendering through reflection and specular effects (see Figure 4.13). Our aim was to determine whether the specular contribution could be learned by our networks. To that end, we selected the best performing model based and view based detectors to test on an object composed of at least two different reflective and specular materials. In our environment, this object was the *Car*.

For both detectors, a cluster radius of 120 and the color descriptor 16 were used. A large radius enabled us to reduce the computational load and, thus, perform testing quickly. In fact, descriptor 16 was one of the best descriptors for the model based approach, but not for the view based approach (see Table 4.1 and 5.3), given the cluster radius and level of realism analyzed. Descriptor 16 included Saturation and Hue components (see Figure 5.11), which are known to be related to specularity [134]. For colors, the best model based detector on LOR1 was the one using the *Bhattacharyya* distance for histogram matching. Likewise, the best view based color detector turned out to be a SOM with a 10×10 neuron grid. Both detectors were trained on the same training set, composed of 170 images similar to the ones shown in Figure 5.18, row (c). The two detectors also shared the same test set, composed of 73 images similar to the ones depicted in Figure 5.18, row (b). As it can be observed from Figure 5.18 row (b), the target object (the Car) in the test set, lacks the specular component, while still retaining reflections (this is particularly noticeable on the glass of the car). The bar graph in the figure shows the difference of *prediction error* between the view based and the model based classifier. The prediction error was a measure of dissimilarity between prediction and observation. For the model based classifier, this was the distance

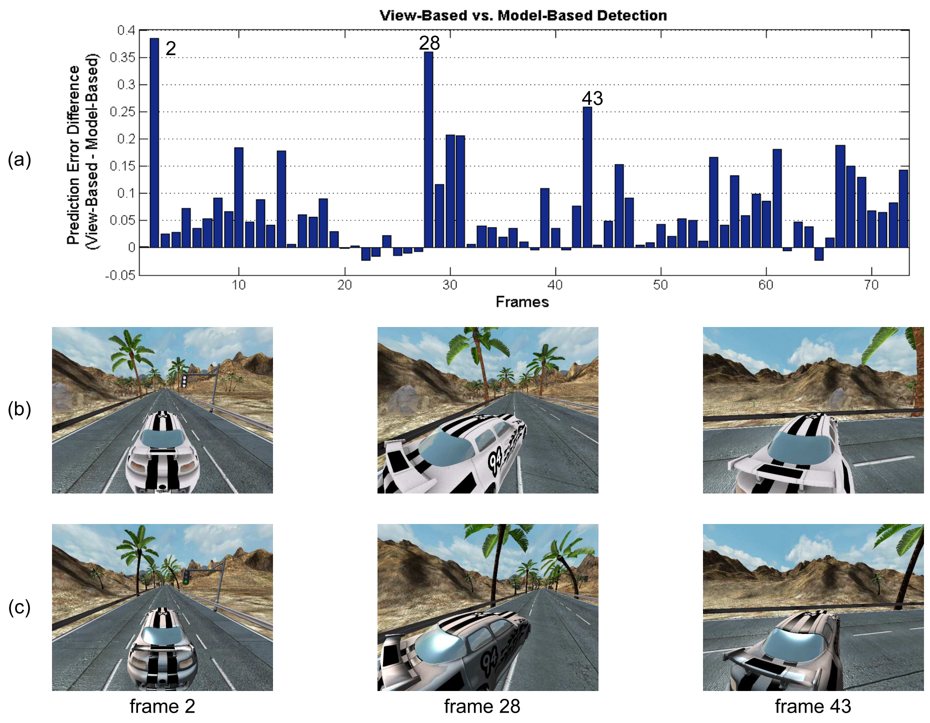


Figure 5.18: Model based vs. view based detection - In order to test the performance of the view based approach versus the model based approach, a context-dependent anomaly was introduced, that is, specularity. The target object (the car) in test images (row (b)) was rendered by suppressing the specular component of light. Both classifiers were trained on a bug-free environment (row (c)). The graphs indicates the difference between prediction errors over a test set of 73 frames. The prediction error was a measure of the difference between system prediction and observation. Because all test images are buggy, high prediction errors are indicators of high accuracy. The greater the error, the more effective the system. This experiment therefore showed that the view based classifier is more suitable for describing complex color dynamics, than the model based approach. Frames 2, 28 and 43 were the ones where the inconsistency was more visible.

between the incoming color histogram and the closest histogram in the database. For the SOM, it was the distance from the observation vector (obtained by combining the transformation matrix color histogram) to the Best Matching Unit (BMU). To compare these two errors, distances were normalized as follows. First, a set of bug-free images similar to the ones from the training set was used to compute the maximum prediction error, for both approaches. Because the images of this additional set were all valid, such an error was treated as a *normalization coefficient*. Next, the prediction error from the both model and view classifier was divided by the normalization coefficient. The normalized prediction error so obtained was used to compare the effectiveness of the two classifiers. Since the test set contained color errors, the prediction error was treated as a effectiveness score for the system; large prediction errors indicated predicted color artifacts of large magnitude.

From the graph it can be noted that the error from the view based classifier is almost always bigger than the one from the model based classifier (the height of the bars indicate the difference between view based error and model based error). This shows that the view based classifier, in this test, is better at detecting context-dependent color artifacts[1].

5.5 Summary

In this chapter we have introduced two connectionist approaches to geometry and color anomaly detection. These are ANN and SOM based classifiers. From the analysis we have performed, both SOM and ANN approaches can be very accurate at detecting geometry issues, upon selecting the appropriate parameter set. As far as color is concerned, however, SOMs are by far more accurate than ANNs, given the types of environment considered and the color bugs reproduced. The color descriptors we used were the ones we used for the model based classifiers, presented in the previous chapter. The shape of the object was described through 2D shape descriptors. As for the model based classifier, here we have assumed the matrix maps and matrix tables are available for training and testing the neural networks.

As with the model based classifier, the target objects contained a considerable amount of noise, that is, occlusion and cluttering. Having noisy training samples was a choice motivated by practical issues. Indeed, in order to remove occlusions, the rendering engine would need to be modified, for instance, by forcing it to render only the objects of interest. Let us assume our estimators need an average of N samples to *generalize* well and that our scene contained M objects. To learn the appearance of the scene, we would then need NM (noiseless) images produced by a rendering system that would need to be modified M times. Because both N and M may well be of the order of hundreds, such a solution appears to be impractical; let alone that removing environment clutter does not rule out self-occlusions. A way of reducing both time and memory resources in collecting training samples is to render as many (possibly noisy) training objects within a single frame. As it turns out, noisy frames cause no major difficulties for the neural networks we used, for they yield accurate and consistent

[1]In our environment, the specularity came from the sun, which was animated. However, the speed the sun was low enough to treat it as a still light source. Hence, the specularity dynamics only depend on the camera-object interaction

results anyway.

In this chapter, we also performed a preliminary comparison of effectiveness between the model based and the view based approach. We showed that, in at least one case, the view based classifier performs better than the model based detector, insofar as what color is concerned. In particular, the view based classifier was better at detecting specularity issues, compared to the model based classifier. By contrast, Table 4.1 shows that a model based approach may be more suitable for detecting geometry artifacts, with respect to a view based approach (see Table 5.2). This seems to be the case given the object analyzed and the geometry descriptors used.

Finally, it is worth noticing that our approach does not require the different light contributions (i.e. ambient, diffuse and specular) to be decoupled and learned separately; that is to say, the color distribution of the object surface is learned without prior information about the optical properties of the object surfaces. Clearly, if the status (e.g. position, color and intensity) of the light sources varies significantly across training and test sets, such information can no longer be factored into the camera-object interaction. In such cases, the light status becomes yet another important piece of information to consider, for color assessment purposes. Neither the model based nor the view based classifiers presented in this work are designed to deal with such complex scenarios; thus, we have not measured the effectiveness of our detectors upon dynamic light conditions. However, we believe that accounting for local illumination effects under the dynamic conditions described above can still be achieved through a view based approach. To this end, the input space of the neural networks could be extended so as to include a description of the light status. With this extra information, the neural networks may be able to learn the light-camera-object dependent color distribution over the object surfaces, much as our current networks effectively learn the camera-object dependent color distribution for static light sources.

Table 5.4: View Based classifiers - ANOVA - ANN geometry tests

	AlphaPalm (OBJ 1)					Windmill (OBJ 2)					Car (OBJ 3)				
Source	Sum Sq.	d.f.	Mean Sq.	F	Prob > F	Sum Sq.	d.f.	Mean Sq.	F	Prob > F	Sum Sq.	d.f.	Mean Sq.	F	Prob > F
rad	0.2121	3	0.07069	8.06	**0**	0.1266	3	0.04220	4.30	**0.0051**	0.0126	3	0.00419	1.31	0.2699
descr	3.4750	14	0.24821	28.30	**0**	2.1351	14	0.15251	15.56	**0**	2.7332	14	0.19523	61.02	**0**
units	0.0089	4	0.00223	0.25	0.907	0.3055	4	0.07637	7.79	**0**	0.0094	4	0.00235	0.74	0.5679
rad*descr	0.4780	42	0.01138	1.30	0.0997	1.0442	42	0.02486	2.54	**0**	0.1103	42	0.00263	0.82	0.7873
rad*units	0.0424	12	0.00353	0.40	0.9629	0.0020	12	0.00017	0.02	1	0.0052	12	0.00043	0.14	0.9998
descr*units	0.2896	56	0.00517	0.59	0.993	0.0310	56	0.00055	0.06	1	0.1454	56	0.00260	0.81	0.8403
rad*descr*units	0.8196	168	0.00488	0.56	1	0.0253	168	0.00015	0.02	1	0.0994	168	0.00059	0.18	1
Error	8.5521	975	0.00877			6.6172	675	0.00980			9.3579	2925	0.00320		
Total	14.3634	1274				10.7753	974				14.5201	3224			

Table 5.5: View Based classifiers - ANOVA - SOM geometry tests

	AlphaPalm (OBJ 1)					Windmill (OBJ 2)					Car (OBJ 3)				
Source	Sum Sq.	d.f.	Mean Sq.	F	Prob > F	Sum Sq.	d.f.	Mean Sq.	F	Prob > F	Sum Sq.	d.f.	Mean Sq.	F	Prob > F
rad	1.4654	3	0.48847	20.40	**0**	0.9807	3	0.32691	25.22	**0**	2.074	3	0.69138	34.52	**0**
descr	4.3668	14	0.31191	13.02	**0**	4.4439	14	0.31742	24.49	**0**	27.034	14	1.93100	96.41	**0**
units	0.5078	4	1.12695	5.30	**0.0003**	0.0634	4	0.01586	1.22	0.2995	0.000	4	0.00005	0.00	1
rad*descr	0.6379	42	0.01519	0.63	0.967	1.5958	42	0.03800	2.93	**0**	3.465	42	0.08251	4.12	**0**
rad*units	0.0061	12	0.00051	0.02	1	0.0740	12	0.00617	0.48	0.9291	0.005	12	0.00045	0.02	1
descr*units	0.0733	56	0.00131	0.05	1	0.2590	56	0.00463	0.36	1	0.010	56	0.00018	0.01	1
rad*descr*units	0.0374	168	0.00022	0.01	1	0.9938	168	0.00592	0.46	1	0.050	168	0.00030	0.01	1
Error	23.3508	975	0.02395			8.7502	675	0.01296			58.586	2925	0.02003		
Total	31.2121	1274				18.0880	974				101.705	3224			

Table 5.6: View Based classifiers - ANOVA - ANN color tests

	Level of Realism 1 (LOR 1)					Level of Realism 2 (LOR 2)					Level of Realism 3 (LOR 3)				
Source	Sum Sq.	d.f.	Mean Sq.	F	Prob > F	Sum Sq.	d.f.	Mean Sq.	F	Prob > F	Sum Sq.	d.f.	Mean Sq.	F	Prob > F
rad	0.02608	3	0.00869	2.96	0.0313	0.04226	3	0.01409	1.90	0.1272	0.1336	3	0.04454	5.91	**0.0005**
descr	0.13723	16	0.00858	2.92	**0.0001**	0.61233	16	0.03827	5.17	**0**	0.0691	16	0.00432	0.57	0.906
units	0.01145	4	0.00286	0.97	0.4206	0.01828	4	0.00457	0.62	0.6501	0.0159	4	0.00397	0.53	0.7162
rad*descr	0.18501	48	0.00385	1.31	0.0769	0.12644	48	0.00263	0.36	1	0.1416	48	0.00295	0.39	0.9999
rad*units	0.02878	12	0.00240	0.82	0.6335	0.02649	12	0.00221	0.30	0.9897	0.0316	12	0.00264	0.35	0.9795
descr*units	0.19809	64	0.00310	1.05	0.3648	0.19864	64	0.00310	0.42	1	0.1529	64	0.00239	0.32	1
rad*descr*units	0.70743	192	0.00368	1.25	0.0156	0.57504	192	0.00299	0.40	1	0.5367	192	0.00028	0.37	1
Error	3.74499	1275	0.00294			7.54635	1020	0.00740			8.9739	1190	0.00754		
Total	5.14558	1614				9.28048	1359				10.1077	1529			

Table 5.7: View Based classifiers - ANOVA - SOM color tests

	Level of Realism 1 (LOR 1)					Level of Realism 2 (LOR 2)					Level of Realism 3 (LOR 3)				
Source	Sum Sq.	d.f.	Mean Sq.	F	Prob > F	Sum Sq.	d.f.	Mean Sq.	F	Prob > F	Sum Sq.	d.f.	Mean Sq.	F	Prob > F
rad	3.0293	3	1.00978	38.76	**0**	4.4691	3	1.48969	22.33	**0**	0.5070	3	0.16899	19.38	**0**
descr	0.6011	16	0.03757	1.44	0.1139	4.2501	16	0.26563	3.98	**0**	0.1206	16	0.00754	0.86	0.6109
units	0.0150	4	0.00376	0.14	0.9655	0.1231	4	0.03077	0.46	0.7642	0.0388	4	0.00970	1.11	0.3489
rad*descr	0.6622	48	0.01380	0.53	0.9967	1.8184	48	0.03788	0.57	0.9924	0.1124	48	0.00234	0.27	1
rad*units	0.0450	12	0.00375	0.14	0.9997	0.0609	12	0.00508	0.08	1	0.0135	12	0.00113	0.13	0.9998
descr*units	0.1080	64	0.00169	0.06	1	0.3979	64	0.00622	0.09	1	0.0585	64	0.00091	0.10	1
rad*descr*units	0.3302	192	0.00172	0.07	1	0.7985	192	0.00416	0.06	1	0.1529	192	0.00080	0.09	1
Error	33.2137	1275	0.02605			68.0458	1020	0.06671			10.3750	1190	0.00872		
Total	38.0545	1614				78.6374	1359				11.5409	1529			

6

Appearance Based Classifiers

The inconsistency detectors introduced in the previous two chapters aimed at modeling consistent behaviour of the virtual environment rendering system, in order to detect visual inconsistencies. We accomplished this task by choosing appropriate feature spaces for representing objects of interest, in order to detect anomalous instances. This approach was quite general, for it did not depend on either the environment or the object to assess, but only on the type of anomaly to detect and the complexity of the virtual environment. We showed how high accuracy can be achieved, even for close-to-realistic environments, under the assumption that the anomaly to detect is directly related to the appearance of the target object, and does not significantly depend on the context in which the object is rendered. Examples of such anomalies are mesh corruption, texture replacement and local illumination artifacts (e.g. wrong or missing specular light components), under static light conditions.

Although such an assumption holds true for a large number of visual defects, there are visual artifacts that are context-dependent in nature. In such cases, consistency becomes a property of the scene, and cannot be ascribed to the atomic objects or events (e.g. the object shape or the camera-to-object interaction). Examples of context-dependent anomalies are reflection, refraction and shadowing defects. In general, the context relates to the status (e.g. position, velocity and material properties) of the objects near the target entity. For instance, the silhouette and contour of shadows depends on the shape and position of the objects interacting with the light, and on the position and intensity of the light sources.

One way to establish context-dependent consistency is to predict what the accurate appearance of the target entity would look like, through a reference version of the software. The prediction from the reference software could then be compared with the

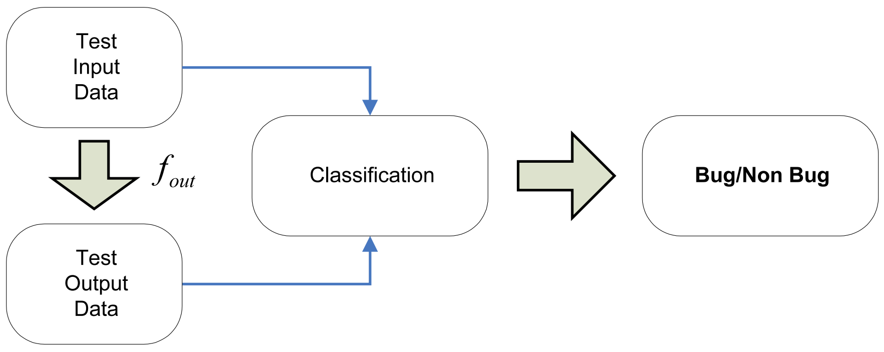
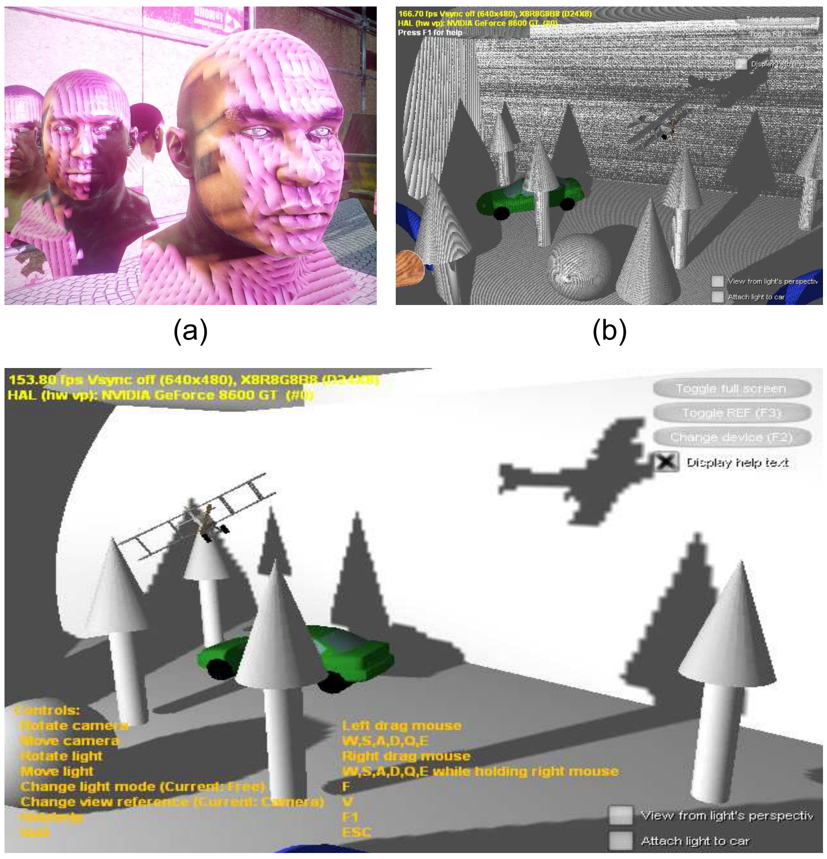


Figure 6.1: Appearance based detection scheme - In an appearance based detection approach, the information about the target anomaly is not inferred from the normal behaviour of the rendering system. Rather, it is encoded in the classifier *a priori*. Because of this, classification can be carried out in one step, by processing the information extracted from the game and searching for the patterns peculiar to the anomaly.

actual output produced by the 3D application for bug detection purposes. This solution, however, would be impractical. On the one hand, the reference software would need to account for a number of visualizations of the same target effect that are plausible given the context at hand[1]. On the other hand, in order for the reference software to reproduce the effect, it would need to access the game context, that is to say, the set of variables used by the game to render the scene. Given that virtual environments are hardly ever designed in such a way as to disclose their internal graphics data, a great effort would be needed from the developer in order to make the application compliant to the autonomous testing device. In this chapter, we will show that when searching for anomalous patterns, it may be easier to infer the inconsistency from the normal rendering of the entire scene. We will show that this is true for at least one case: a shadow aliasing artifact. As we will see, the only information required from the game in order to establish correctness are images containing only shadows. Because the detection is accomplished by analyzing the appearance of the target entity on screen, we shall refer to this approach as *appearance based* detection. The appearance-detection mechanism, depicted in Figure 6.1, has no learning components; the knowledge about the anomalous pattern is encoded in the detector, rather than being inferred from the learned behaviour of the rendering system.

[1]Visual effects are typically constrained to visual realism requirements of the scene. Accordingly, shadows can present soft, realistic edges in photo-realistic environments, or they can be rendered as sharp dark regions, in cartoon-like games.


(a)
(b)
153.80 fps Vsync off (640x480), X8R8G8B8 (D24X8)
HAL (hw vp): NVIDIA GeForce 8600 GT (#0)
Toggle full screen
Toggle REF (F3)
Change device (F2)
Display help text
Controls:
Rotate camera
Left drag mouse
Move camera
W,S,A,D,Q,E
Rotate light
Right drag mouse
Move light
W,S,A,D,Q,E while holding right mouse
Change light mode (Current: Free)
F
Change view reference (Current: Camera)
V
Hide help
F1
Quit
ESC
View from light's perspectiv
Attach light to car

(c)

Figure 6.2: Common shadow anomalies - The realism of shadow techniques is often determined by various parameters of the shadowing algorithm. If the parameters are not properly configured, unintended artifacts may occur, such as *projective aliasing* (a), shadow *acne* (b) and *perspective aliasing* (c).

6.1 A Shadow Aliasing Detector

Shadowing plays an important role in the realism of the scene, for it helps the viewer to determine spatial relationships between the geometries of the environment. Shadows are appealing visual effects that a virtual environment should possess, as shadowing is likely to promote *immersion* [66]. Over the years, a number of fast approaches to generating realistic shadows have been proposed. A good survey of them can be found in [56] and [125]. However, each one of these techniques has known issues that, if neglected by the designer, may lead to visually unappealing renderings (Figure 6.2).

As mentioned earlier, shadowing is a context-dependent visual effect. We wish to show that, although the correct shadowing behaviour may be hard to formalize, simple and effective formalizations of shadow artifacts are possible. We will see this on a specific case concerning one popular algorithm known as *shadow mapping* [124].

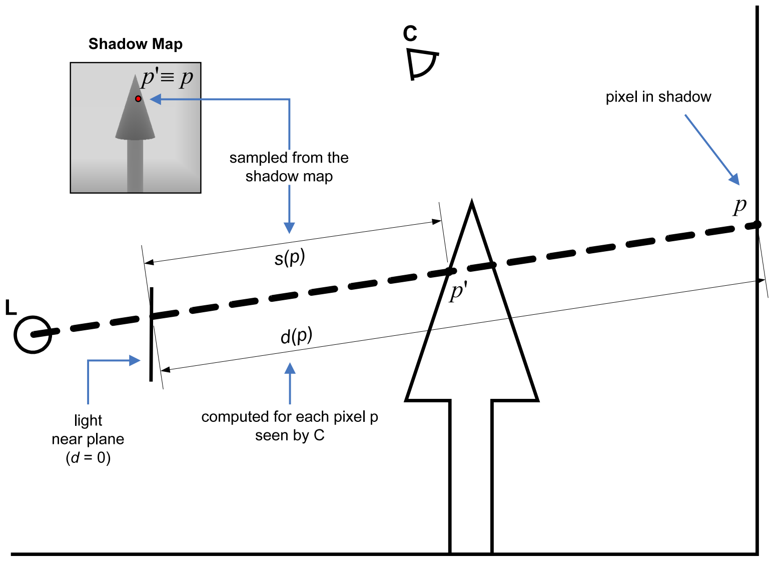

Shadow Map
$p' \equiv p$
C
pixel in shadow
sampled from the shadow map
p
s(p)
p'
L
d(p)
light near plane (d = 0)
computed for each pixel p seen by C

Figure 6.3: Illustration of the Shadow Mapping algorithm - For each pixel p seen by the camera C, its distance $d(p)$ from the light source L is computed. If $d(p)$ is bigger than the distance $s(p)$ of the pixel p' closest to the light along the line of sight from the position of the spotlight to p, then p is considered in shadow. The position of the pixel p in the shadow map is determined by transforming the coordinates of p to *light-space* coordinates. This is achieved through the camera-to-light transformation matrix, derived from the light and camera position, and direction.

6.1.1 Background: The Shadow Mapping Technique

Shadow Mapping is a commonly used technique for generating realistic shadows in arbitrarily complex environments. Its efficiency and versatility makes this algorithm the preferred shadowing mechanism in the film and computer games industry [146]. The approach consists of rendering the scene twice. One time from the viewpoint of the light and a second time from the camera viewpoint. In the first rendering, the light-pixel distance (depth) of each pixel visible from the light source is written into the so called *shadow map*, a square matrix of predefined size or resolution. The generation of shadows takes place at the second rendering stage. To that end, the distance light-pixel of each pixel seen by the camera is compared with the distance written in the shadow map. If the shadow map value of the pixel is smaller than the distance just measured, then the pixel is considered in shadow. Indeed, a smaller pixel depth in the shadow map indicates that the pixel currently observed by the camera is occluded by some surface closer to the light. Because the current pixel is occluded by some other object, it will need to be darkened by the rendering system. Figure 6.3 illustrates this idea.

Using low resolution shadow maps is a choice motivated by computational constraints. Intuitively, the higher the shadow map resolution, the larger the amount of pixels that will be processed by the shadow map algorithm, the lower the framerate of the rendering system and the 3D application in general. Over the years, a number of approaches have been presented in order to reduce shadow map aliasing, via eliminating the cause of the issue; that is, the distortion of the shadow map texels (texture pixels) due to perspective transformations [79]. The approach presented in this chapter, aims at determining whether and where the aliasing effect becomes visible on screen, regardless of the strategy used for synthesizing the shadows itself.

This approach, being an *image space* technique[1] has the advantage of efficiency, for it will not process the geometric information of the environment but only the projection of the geometries on a 2D support — either the shadow map or the final image. However, like most image space techniques, the shadow mapping algorithm is prone to aliasing artifacts [146]. The visual anomaly we are to define and detect in this chapter is referred to as *perspective aliasing.* Such an artifact, depicted in Figure 6.2 (c), consists of blocky shapes that the shadow forms over the geometry to which it is projected. The phenomenon occurs when a low resolution shadow map is used and it is particularly visible in those regions where the distance from the camera is far less than the distance from the light source.

6.2 Problem Formulation

We define a region of the screen to be affected by shadow aliasing if the region has all of the following properties:

- it is a segment of an edge of a shadow silhouette;
- it forms at least p consecutive corners, displaced at a close distance from one another;
- it does not contain excessively long straight segments, compared to the size of the corners.

[1]Image space techniques refer to those Computer Graphics approaches that aim at processing the 2D information resulting from the projection of 3D objects to screen. By contrast, objects space techniques base their functionality on the 3D object space information.

These properties should hold true, no matter the size, location or orientation of the aliasing effect. In other words, we want this description to be independent of affine transformations.

To build such a region detector, we first note that an aliased shadow silhouette appears as a sequence of nearby corners of different size and shape (Figure 6.2 (c)). Thus, one feature our detector should have is the capability of identify corner structures. In image processing, robust corner detection techniques exists. Most of them are based on a multi-scale descriptors of image structure, known as a *second moment matrix* [99]. We can define such a descriptor[1] $\mu : \mathbb{R}^2 \rightarrow SPSD(2)$ by

$$\mu_L(q, \sigma_I, \sigma_D) = \begin{pmatrix} \mu_{11} & \mu_{12} \\ \mu_{21} & \mu_{22} \end{pmatrix} = E_q(\sigma_I) \begin{pmatrix} L_x^2(\sigma_D) & L_x L_y(\sigma_D) \\ L_x L_y(\sigma_D) & L_y^2(\sigma_D) \end{pmatrix} \tag{6.1}$$

where $E_q(\sigma_I)$ denotes the spatial window (a Gaussian kernel) centered at $q = (x, y)^T \in \mathbb{R}^2$ and of size σ_I (the integration scale), used for accumulating statistics of the pointwise descriptor. The term L_a is the derivative of the image computed in the a direction. The matrix, in effect, describes the gradient distribution in a local neighbourhood of a point. The local derivatives are computed with Gaussian kernels of size σ_D (the local scale). The eigenvalues of such a matrix represent the two principal signal changes in the neighbourhood of a point. Thus, if both curvatures are significant, that is, the signal change is significant in both orthogonal directions, we have detected a corner. Figure 6.5 (b) shows an example of regions (gray areas) where both curvatures are significant.

Identifying corners in the image, however, is not sufficient to determine whether the anomaly is actually present or not. On the one hand, nearby corners of shadow silhouettes may belong to different objects. If we did not allow for this possibility, many false alarms (false positives) could be raised. Moreover, real (or realistic) images typically contain a large number of corner structures formed by texture gradients, object contours or by the occlusion of the objects silhouette by other objects. Therefore, if the region of interest (i.e. shadows) was not enhanced in some way, even a robust corner detector will fail to find the corners of interest as it will likely be "confused" by more prominent — though irrelevant — corner structures (Figure 6.4). Enhancing the regions of interest in the image at hand has a very simple solution in our case. As will be discussed in the next chapter, it is possible to extract images containing shadows only from the graphics pipeline. If only shadow information is presented to the algorithm,

[1]The notation SPSD(2) stands for the cone of Symmetric Positive Semidefinite 2×2 matrix.

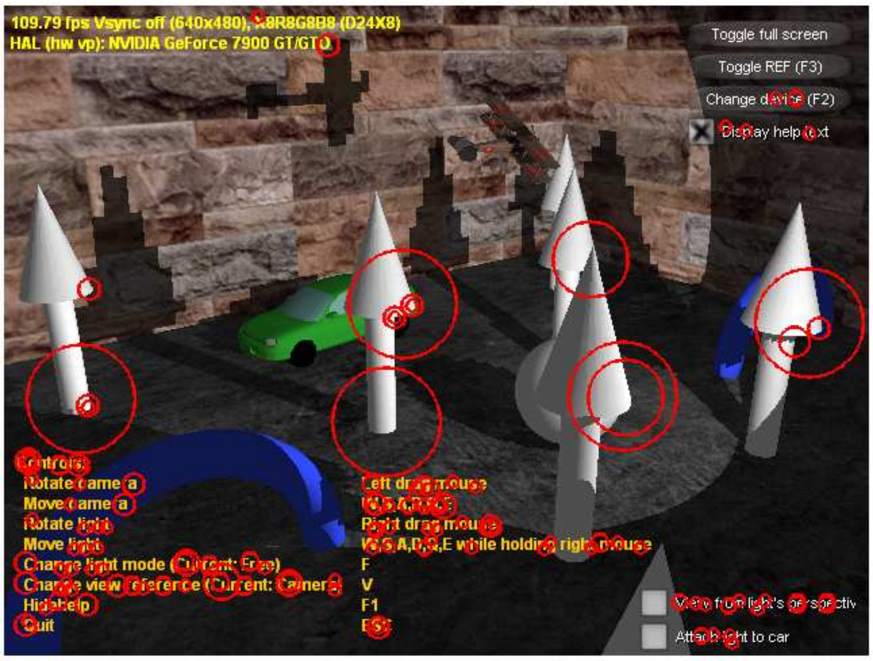


Figure 6.4: Example of corner detection in a synthetic image - The image shows an example of a virtual environment affected by shadow perspective aliasing. Although the jagged shape of shadow edges is quite evident to a human observer, the robust corner detector algorithm (Harris-Laplace) determined that the most significant corner structures (denoted by red circles) are elsewhere in the image.

then it is more likely that corner aggregates will form in relevant anomalous image regions.

Another feature our algorithm should posses is the ability to determine whether two corners belong to the same shadow edge or not. To achieve this, we first need to determine where the shadow edges are in the image. In our solution, we used the Canny edge detection algorithm [20] applied to *shadow-only* images (Figure 6.5 (c)). The term shadow-only is used here to denote images containing only shadows. Combining the corner detector with the edge detector, we can identify aliasing artifacts, as we will explain in the next section.

6.2.1 Implementation

Our solution to shadow aliasing detection contains the following steps:

1. determine regions where corners are densely packed (corner regions), and identify corner points;

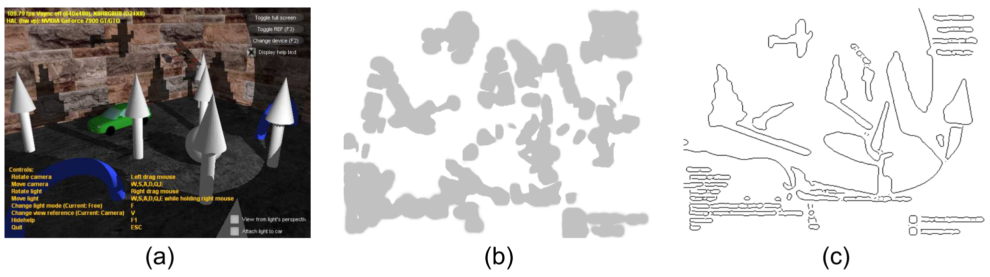


Figure 6.5: Potentially anomalous shadow regions - The detection of potential shadow defects concerns two processes. These are the identification of regions where corners are most densely packed (b), and the extraction of shadow edges. The extraction of corner regions and shadow edges helps in reducing the likelihood of missclassifying shadow aliasing regions.

2. extract the set of shadow edges;
3. remove from the edge set those segments that do lie within the corner regions (i.e. straight lines);
4. remove from the edge set those segments that do not present at least p corners (where p is a parameter);
5. report the remaining segments (aliasing regions).

To determine corner regions and corner points we used the Harris response R [53]. This is an image descriptor which combines the trace and the determinant of the second moment matrix μ introduced earlier:

$$R = \det(\mu) - k\,\mathrm{trace}^2(\mu) \tag{6.2}$$

where $k > 0$ is a tunable sensitivity parameter. Equation 6.2 does not involve computing the eigenvalues of μ but only its determinant and trace; this makes it computationally attractive. This *cornerness* measure R presents hills in correspondence to corners and valleys in correspondence to edges. Hence, by applying a threshold τ to R we can generate a binary image of blob-like regions, which we call the *corner region image* (Figure 6.6); such blobs originate from the (downwards) projection of R onto the $z = \tau$ plane. Thus, corner regions denotes the neighborhood of a specific Harris corner point. Since the diameter of the corner regions depends on the size σ_I of E_q, we can modify σ_I so as to make nearby corners aggregate into a single region.

As stated earlier, shadow edges were extracted through the Canny edge detector algorithm. All Canny segments passing through a corner region are retained as they

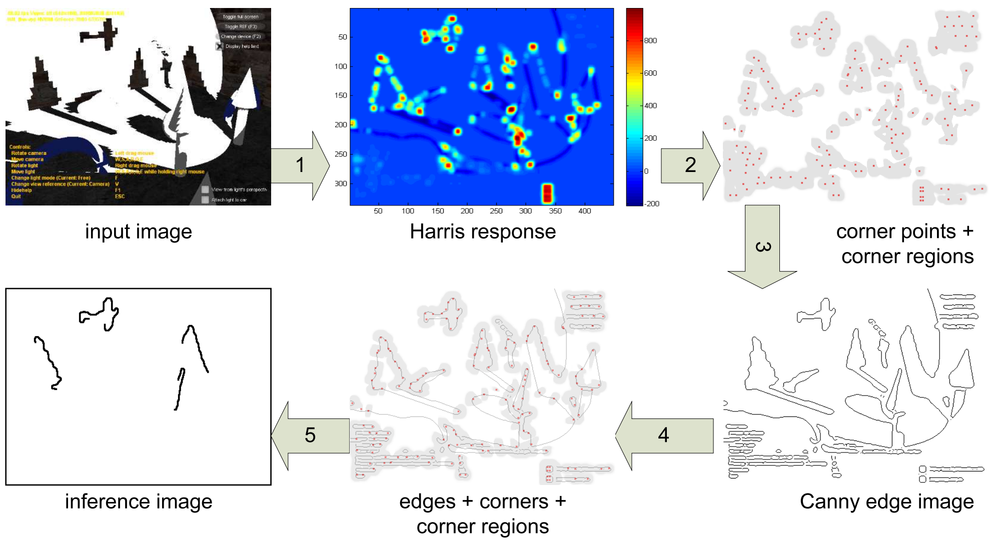

1
2
3
4
5
input image
Harris response
corner points + corner regions
Canny edge image
edges + corners + corner regions
inference image

Figure 6.6: Shadow map aliasing detection - The first stage of aliasing detection consists in computing the Harris response from the shadow-only image extracted from the game. The Harris response peaks in the neighbourhood of corner structures and presents valleys (of negative value) along edges. Next, a threshold is applied to the Harris response in order to identify potentially buggy regions (corner regions). Also, the corner points are determined at this stage, by computing the position of the maxima of the Harris response. Next, edges are extracted through the Canny edge detector algorithm. Finally, all corner region are assessed against aliasing. An edge lying within within a region is considered alias if it bends through at least p corner points (6 in this example).

represent corners. All other Canny segments are discarded, for they are straight lines. To ensure that the selected segments belong to a line bending through at least p corners, we simply count the nearby corners of a segment. The minimum size of consecutive corners can be parameterized through σ_I: the bigger the filter, the bigger the size of the minimum jagged pattern that will be recognized as anomalous. In fact, in order to detect artifacts of different magnitudes and to make the algorithm scale-independent, the second moment matrix was computed at different scales. Figure 6.6 shows the process of extracting aliasing segments from an image containing shadows only. Our shadow aliasing detection solution is listed in Algorithm 6.1.

After defining an appropriate array in integration scales, the RGB input image is converted into a black and white image through the function Rgb2Bw, which performs an RBG-to-gray scale conversion, followed by an adaptive thresholding. Next, the edges are extracted from the black and white image through the function Canny, which performs a Canny edge detection through a Gaussian filter of the same size as the minimum of the integration scales. The process iterates through the array of integra-

Algorithm 6.1: Shadow Aliasing Detector

```
1  function AliasingDetector($\mathbf{I}$,$\tau$,$p$) returns a binary matrix
     inputs:
           matrix $I$ denoting the frame to analyze, of size $m \times n$
           real $\tau$ denoting the threshold to apply to the Harris response
           integer $p$ denoting the min number of corners of a jagged edge
     locals :
           array iScales of integration scales $\sigma_I$, of size $1 \times s$
           binary matrix bwImg, of size $m \times n$
           binary matrix cannyImg of Canny edges, of size $m \times n$
           reals sigmaD and sigmaI denoting local and integration scales, respectively
           real matrix harrisResp of Harris responses, of size $m \times n$
           binary matrix harrisCorn of Harris points, of size $m \times n$
           binary matrix blobImg of corners neighbourhood, of size $m \times n$
           binary matrix aliasImg of detected jagged edges, of size $m \times n$, initially zero
2    iScales ← array of $s$ integration scales, e.g. $[1, 2, ..., s]$
3    bwImg ← Rgb2Bw($I$)
4    cannyImg ← Canny(bwImg, Min(iScales))
5    foreach $i$ in iScales do
6        sigmaD ← $2i$
7        sigmaI ← $i$
8        harrisResp ← Harris(bwImg,sigmaD,sigmaI)
9        harrisCorn ← Peaks(harrisResp)
10       blobImg ← Threshold(harrisResp,$\tau$)
11       aliasImg ← OR(aliasImg,Filter(cannyImg,blobImg,harrisCorn,$p$))
12   end
13   return aliasImg
```

tion scales, adding — via the boolean operator OR— the new jagged Canny segments detected to the final matrix aliasImg. A jagged line is detected by first computing the Harris response as per Equation 6.2, through the function Harris. As suggested by Mikolajczyk [99], local and integration scales should be in the relation $\sigma_D = s\sigma_I$, where s is a constant factor, set to 2 in our experiments. The corners are then extracted by Peaks which returns a binary matrix of 1s at the location of the local maxima or the Harris response, and 0 elsewhere. To compute the corner points neighbourhood, a positive threshold τ is applied to the Harris response, through the function Threshold. After the thresholding operation, the (i, j) element of the matrix blobImg will be 1 if harrisResp$[i, j] > \tau$; and 0 otherwise. The higher the threshold, the smaller the corner regions, as the $z = \tau$ plane will intersect the Harris function at its peaks. On the other hand, a too low threshold may intercept R at its *plateau* (where the function is equally zero), in which case, almost the entire image will become a corner region. We found that $\tau = 1$ produces large enough corner regions, in most cases. Finally, the output matrix aliasImg is updated by adding, through an OR operator, the Canny segments that exhibit both the following properties: lie within some corner region (the 1s of the matrix blobImg) and bend through at least p corner points where p is a parameter, set to 6 in our experiments. The matrix of such jagged Canny edges is produced by the function FILTER. The non-zero entry of such a matrix denotes the regions of the screen where the aliasing anomaly is present.

6.3 Results

In order to test our solution, we used a simple environment which implemented the shadow mapping algorithm. Such a test-bed was one of the samples available from the Microsoft DirectX® 10 Software Development Toolkit[1]. In our experiments we introduced shadow aliasing by decreasing the resolution of the shadow map to 112×112 pixels; this resolution was low enough to produce visible shadow aliasing artifacts. Our algorithm was applied to the images containing only shadows, rather than the final frames. These shadow-only images were manually extracted from the graphics pipeline through the *shader debugger*[2] Microsoft Pix®[3]. The automatic extraction of shadow-only images, however, is a process that can be performed automatically, as will be explained in the next chapter.

The results of our solution can be observed in Figure 6.7. The original output from the game is shown in column (a); column (b) is the related shadow-only image; and column (c) represents the output from our classifier. White lines in the output image denote regions identified as anomalous by our algorithm. Note, that in the shadow-only frames we included the text from the original frame. We did so, to show the robustness of the algorithm to irrelevant data (i.e. noise). Row 1 and 3 in the figure, are two renderings of the same environment from two different camera angles. Here, the scene is affected by shadow perspective aliasing. Rows 2 and 4 are the bug-free visualizations of scenarios 1 and 3 respectively. To correct the anomaly, a higher resolution shadow map (512×512 pixels) was used.

To test the robustness of the algorithm to affine transformations, we generated staircase structures of different size, position and orientation. As can be observed from

[1]http://www.microsoft.com/

[2]Shader debuggers are tools that emulate the graphics pipeline in order to facilitate the debugging of graphics applications. Through shader debuggers, the rendering process can be interrupted and the graphics data extracted for assessment.

[3]http://msdn.microsoft.com/en-us/library/

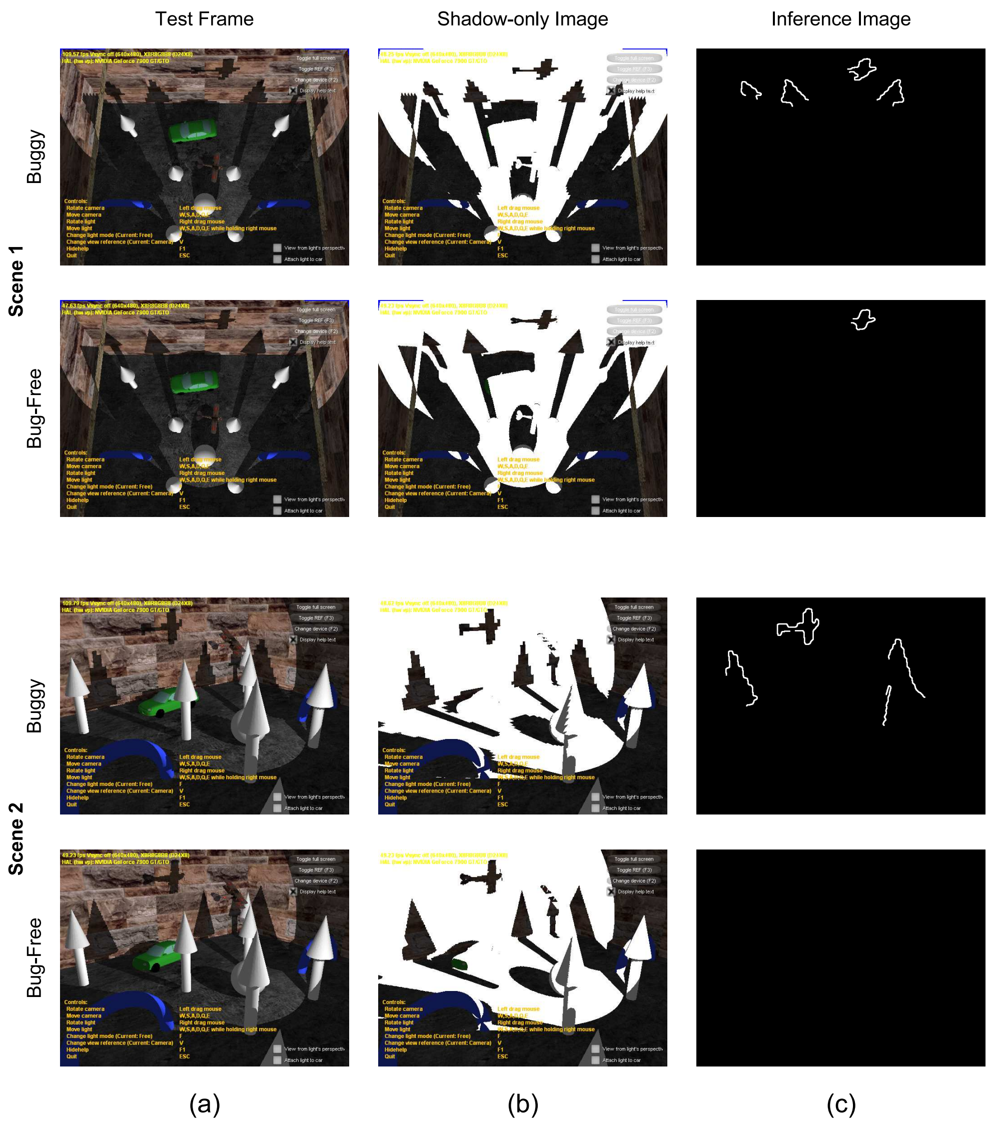


Figure 6.7: Shadow Aliasing detection - Rows show a buggy and related bug free version of the same scene. In particular, the shadow aliasing effect is present in rows 1 and 3, column (a). The shadow-only version of the scene is shown in column (b). Column (c) shows how blocky shadows are detected (white lines) by the algorithm.

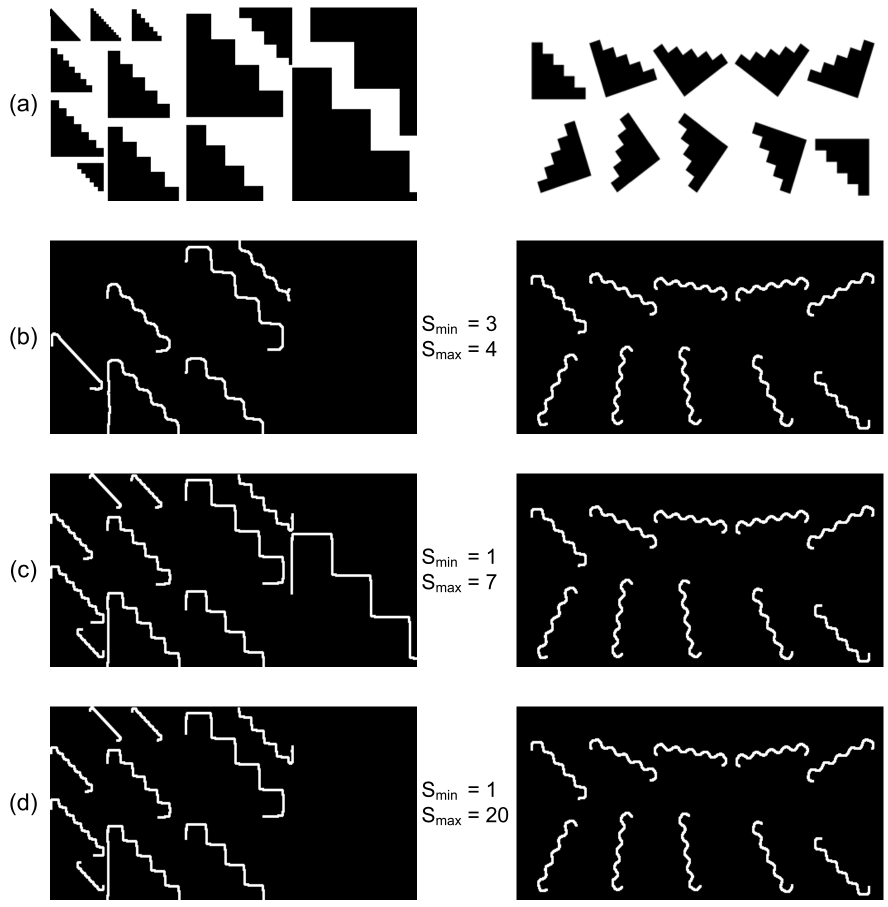


Figure 6.8: Robustness to affine transformations - The appearance based detector presented here proves to be robust to affine transformations. The minimum and maximum aliasing magnitude can be controlled through minimum and maximum integration scales. Row (a) shows the input to the aliasing detector algorithm. The outputs presented in rows (b), (c) and (d) have been produced via (minimum, maximum) integration scales of $(3, 4)$, $(1, 7)$ and $(1, 20)$ respectively. Decreasing the integration scale enables the detection of smaller magnitude aliasing. Similarly, increasing the maximum integration scale allows the detection of bigger staircases up to a threshold. Increasing the maximum integration scale over a value of 7 will not increase the maximum aliasing size that can be detected. This can be observed by comparing the results of rows (c) and (d), column 1. In row (c) a maximum integration scale of 7 was used; the output of row (d) was produced by using a maximum integration scale of 20. Note how the solution (c) allows the detection of an artifact that is not detected by solution (d). In fact, any maximum integration scale from 8 to 20 produces results (not shown in the picture) similar to the one of row (d). This suggests that 7 is a good maximum integration scale value.

Figure 6.8, almost all jagged edges were correctly detected. Note also how straight segments belonging to the staircase silhouettes are correctly filtered out by the algorithm. The figure also shows the flexibility offered by the algorithm regarding the minimum and maximum aliasing magnitude that can be detected. In particular, it can be seen that decreasing the minimum integration scale allows the detection of smaller aliasing effects; similarly, a bigger maximum integration scale enables the detection of more severe issues. We noticed a peculiar behaviour of the detector, however, that is worth reporting here. There seems to be a threshold on the maximum integration scale, above which the performance of the algorithm seems not to improve or even deteriorate. The results (b), (c) and (d) in the first column of Figure 6.8, shows that an increase of the maximum integration scale allows the detection of bigger aliasing effects (i.e. bigger staircases), as expected. However, there is a limit on the maximum jag size that can be detected and thus, on the maximum integration scale that is to be used. Increasing the maximum integration scale above a threshold (7 in our experiments) will not translate to detecting more prominent aliasing defects; in fact, it will prevent the algorithm from detecting as many defects as the solution that sets the maximum integration scale to the threshold value.

6.4 Summary

In this chapter we have presented an appearance based mechanism to be used for detecting shadow aliasing issues. Shadow aliasing is an example of anomaly affecting a context-dependent visual effect. Defining the normal behaviour of context-dependent effects for debugging purposes can be a hard task; indeed, the description of the expected visualization would imply allowing for the context (i.e. game status) in which the effect appears. The task would become even harder if we wished to allow for different plausible (i.e. visually appealing) renderings of the same effect within the same context. We have shown that, if we look at this problem from a different perspective, fairly simple and effective detectors can be designed. We have shown this in one specific case, that is, the detection of shadow aliasing artifacts produced by the shadow mapping algorithm.

The solution we proposed is both simple and robust and is based on *a priori* knowledge of the anomalous pattern to recognize. Insofar as the shadow mapping algorithm is concerned, our system could potentially be used as a tool for determining when and where to increase the resolution of the shadow map. The method is not general, however, as different context-dependent anomalies are likely to exhibit different patterns.

Nevertheless, given the simplicity and effectiveness of the algorithm, and given the very limited amount of data required from the game, the appearance based mechanisms can prove to be a valid approach. Indeed, it can represent a *local* alternative to the hard task of generating a *general* model of the ideal, bug-free rendering system.

The algorithm presented in this chapter uses some internal information from the game. This is the *shadow-only* image, that is, an image containing only shadows. As we shall see in the next section, such data can be automatically extracted from the graphics pipeline. Such a process is different from the model based and view based schemes, as the appearance based approach does not process any geometry information (i.e. transformation matrices and 3D geometries). Instead, inference is only based on what appears on screen, versus what it is expected to appear. Hence the term, appearance based detection.

7

Towards the Autonomous Retrieval of Debugging Data

In the previous chapters we have presented various ways of assessing virtual environments against visual defects. In all cases, we have implicitly assumed that an interaction between the testing machine and the environment to test needs to be established. The testing machine will send stimuli to the environment to test; in return, the environment will produce outputs that will be analyzed by the testing machine. Regardless of the type of information transferred between the two mechanisms, it is important that the system to test receives "reasonable" stimuli and that the data exchange process does not compromise the functioning of the environment itself. Indeed, as we noted in early chapters (Chapter 1 and 2), an ideal anomaly detector system is a device that explores and tests virtual environments in a way that is indistinguishable — for all testing purposes — from the way a human would do. Moreover, the behaviour of the environment to test should not depend on the existence of a testing device (observer) and on what the device is doing with the acquired data. If the testing device influences the output of the system to test, the debugging process becomes a meaningless exercise; the inference will be made on data that would not be observed under normal circumstances. This latter observation motivates a key debugging system feature; the automatic testing of any system should not require extensive modifications to the software and/or hardware to test. If the testing context is significantly different from the normal scenario, then one should expect the testing output to be different from the normal play.

This work focuses on visual consistency issues, hence the stream of testing data we refer to is the graphics information used by the application to render the scene. Either the model based, the view based or the appearance based detectors presented earlier

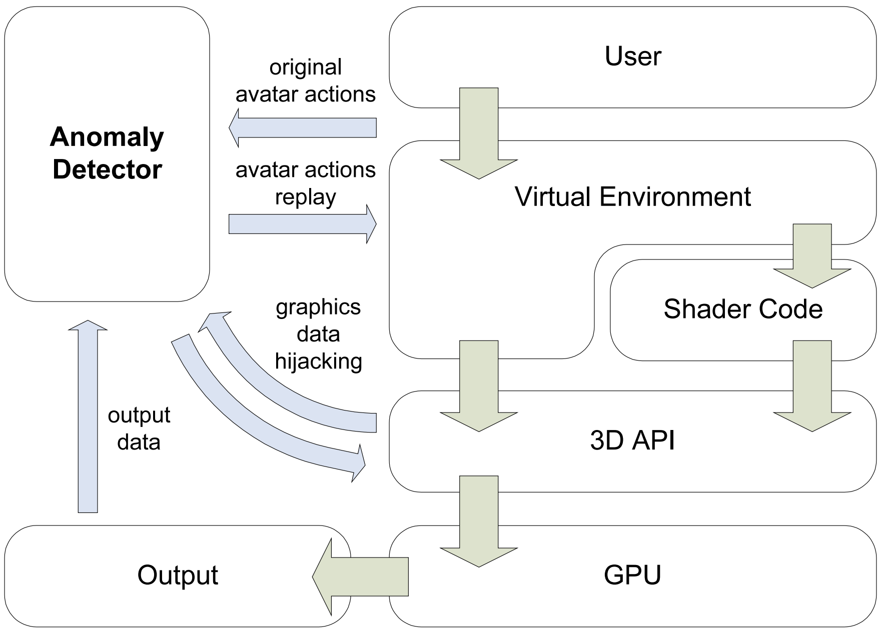


Figure 7.1: Debugging System - Virtual Environment Interaction - Virtual Environments can be assessed against visual inconsistencies using environment-independent anomaly detectors. To that end, both virtual environment and detector should communicate through input and output test interfaces. In this way, human-like input can be generated by the testing device and the output graphics data extracted. The graphics data can be "hijacked" from the graphics drivers in order to retrieve and modify graphics calls and data (e.g. meshes and shader code) directed to the GPU. Because inference will eventually be made on the final images produced by the game, the anomaly detector will also need to retrieve the output (frame) buffers from the GPU.

process data extracted through appropriate modification to the original source code of the game engine. In this chapter, we discuss ways of extracting input and output information from any virtual environment, without modifying the environment itself. The general architecture we will refer to is depicted in Figure 7.1. The anomaly detector will have test interfaces towards the input to, and output from, the 3D application. In particular, it will need to collect the input data stream from human play sessions in order to replicate it later on for debugging. In order to limit the interfere to the application to test, the graphics data will need to be intercepted at the driver level, before being sent to the GPU. Querying the graphics drivers will enable the application-independent retrieval of the graphics APIs (Application Programming Interface) currently used by the application to render the scene, as well as the data that will be processed by the graphics hardware (GPU); this includes textures, vertex and index buffers, and shader code. In fact, in order to produce extra test data (such as the matrix maps introduced in Chapter 4) the graphics stream could be “hijacked” to the anomaly detector, modified and eventually sent to the GPU for processing. Finally, the final output (images) from the game will need to be extracted from the graphics hardware in such a way as to make inference about the image that is perceived by the user.

7.1 Human-Like Action Acquisition

The strategy we adopted for traversing the environment is quite simple. In order to make the game produce data that would be generated by a human play-session, we recorded the play sessions from human players and store only the information related to the *avatar*, that is, the character or object directly controlled by the user. In our case, the avatar-related information was stored through an *action table* containing the *world* matrices relative to the avatar (car) of the (racing) game we used, along with the environment position (track number) at which such matrices were sampled. The action table is a one-dimensional *lookup* table whose values encode the position-dependent user actions. In order not to significantly slow down the 3D game, the sampling rate at which the matrices were sampled was kept low (about one matrix every two seconds). Both training and testing of the anomaly detectors was performed on data that were generated via re-playing the user actions through the action table. Whenever the game called the update function f_{tr} (see Chapter 1), the current track position was computed and the closest track numbers in the table used to determine the appropriate world matrix to employ in order to move the avatar (Figure 7.2). This solution produced consistent avatar dynamics, given the sample frequency we used.
In the vast majority of virtual environments, the path the camera follows is highly influenced by where the avatar is and what it is doing. By stimulating the game with the avatar’s actions collected from a human-play session, we ensured that the camera always followed human-like paths.

In order to re-play the avatar action sequence, the experimental game engine was modified and the avatar-related transformation matrices extracted. However, this type of data can be automatically acquired via querying the graphics drivers. In modern computer games, the geometry of virtual objects is transformed by vertex shaders (see Chapter 4 for more details). To that end, vertices and related transformation matrices are passed to the shader processors through graphics calls, as shown in Algorithm 7.1 (line 3 and 4). If the vertex structure (mesh) can be identified in the code, then the related matrix can be read from the drivers, at the memory location indicated by the argument of the related graphics call.

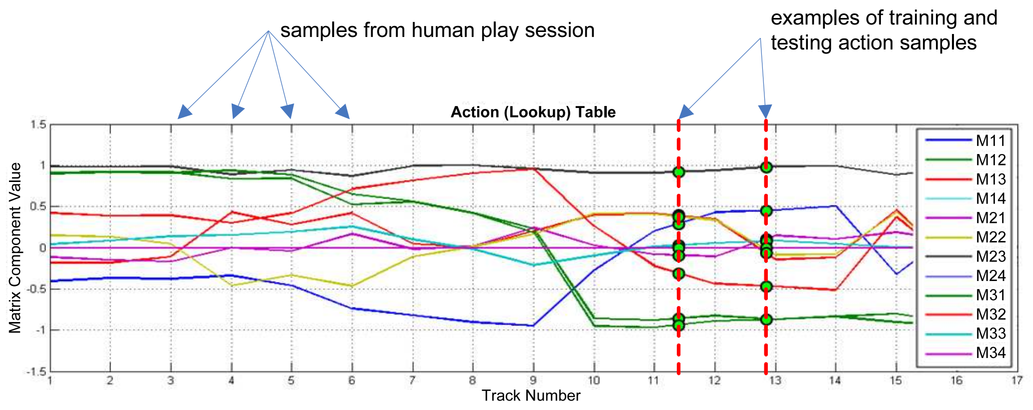


Figure 7.2: Avatar Action Acquisition - In this work, the action sequence performed by the human player is represented through the transformation matrices applied by the game to the avatar (car). In our experiments, the *world* matrix (i.e. the matrix transforming the avatar geometry and pose from local space to world space) was used. Since the world matrix is generated by the game upon user commands (actions), such a matrix encodes the activity of the human player. World matrices where recorded during a normal play session and linked to *track numbers* via a lookup table. The track numbers identified the position of the avatar in the world. Adjacent values in the table were linearly interpolated. In this diagram, only the first 12 values $(M_{11}, M_{M_{12}}, \ldots, M_{33}, M_{34})$ of the world matrix are represented. This mechanism enables a mapping from environment position to avatar pose. In particular, human-like action sequences can be reproduced by simply querying the table at the appropriate environment position. In this work, the re-played actions were used for training and testing the classifiers.

This mechanism of action re-play through a transformation matrix representation has proven useful for the research work presented here. If, however, we wish to endow the testing *agent* with autonomous navigation capabilities and introduce plausible variety in the action sequences, more complex mechanisms are needed. An autonomous artificial agent should be capable of acquiring navigation and interaction skills by observing the human play[1]. Research in this direction has been carried out in recent years. As an example, we cite the work conducted by Thurau [136], who showed how the human-play related network traffic can be used to build artificial agents that learn how to play (first person shooter) games by observing human-play sessions. This approach assumed that the status of the virtual characters (e.g. position, direction, velocity and action performed) is available at any time. Disclosing information of this kind is, in fact, a design requirement for most computer games that feature multiplay experiences. Clearly, the more data is available from the game about the environment, the easier becomes the building of sophisticated models of human behaviour.

[1]Training is required if limited or no prior knowledge of the environment is available.

7.2 Graphics Data Extraction and Modification

Although, games do not typically produce the extra data we used for training and testing the classifiers[1], graphics data can still be automatically extracted from the rendering system and modified in real-time. This can be achieved by intercepting the data stream from the application at the drivers level, applying the intended modifications, then sending the data back to the GPU for ordinary rendering (Figure 7.1). Access to the graphic data mainly depends on the Application Programming Interface (API) used by the virtual environment to control the graphic hardware. The most popular interfaces used by modern graphics applications (including computer games) are OpenGL®[2] and Microsoft DirectX®[3].

In OpenGL based applications, the data can be re-directed by replacing the OpenGL driver with one that can *hook* the 3D calls via a locally installed Dynamic Link Library (DLL) [101]. Since the linking of the library happens dynamically, when a shared library is replaced, the next invocation of a program that uses the library will operate with the updated version. This mechanism allows the modification of the rendering behaviour of the application without altering the application itself. The command stream intercepted through low-level graphics APIs, represents a *programmatic* representation of the geometries of the scene: it provides a sequence of commands that, when executed in order, draw a picture of the geometry.

Extracting data for DirectX drivers is generally more complicated than in OpenGL. DirectX is based on the Component Object Model (COM), which define classes inheriting from the DirectX interfaces. When a 3D application invokes the DirectX library to create an instance of interface, the "hijacking" library should intercept the graphics calls and invoke the related functions in the original DirectX library to create an original instance of interface. Pan et al. [108] followed this approach for intercepting DirectX calls. The authors designed the library in such a way as to define image classes according to the DirectX interfaces. After intercepting the graphics commands, the library could invoke the appropriate function of the DirectX library on behalf of the game application in such a way as to control, modify and maintain the interface instance. Mechanisms similar to the ones used for OpenGL, however, may also be used for intercepting and modifying the DirectX data stream, through *proxy DLLs* [71]. To

[1]For the model based and view based approaches the extra graphics data used were the matrix maps. By contrast, the appearance based classifier processed images containing only shadows.

[2]http://www.opengl.org/

[3]http://www.microsoft.com/games/en-au/aboutgfw/pages/directx.aspx (June 2011)

that end, the proxy DLL should be loaded by the game on its start-up; such a library would then pass the (modified) 3D commands to the original DirectX library.

Tools based on this data re-direction principles are available for both DirectX and OpenGL. Such tools are known as *shader debuggers*, for they enable the developer to debug the programmable part (shaders) of the graphics pipeline. DirectX data can be intercepted through Microsoft PIX®, bundled with the Microsoft DirectX Software Development Kit (SDK)[1]; and 3D Ripper DX®[2]. The OpenGL drivers can be hooked through GLIntercept®[3], an open source application. Since our test bench environments used DirectX, we will focus on automatic graphics data interception through Microsoft PIX.

7.2.1 Intercepting Training and Testing Data

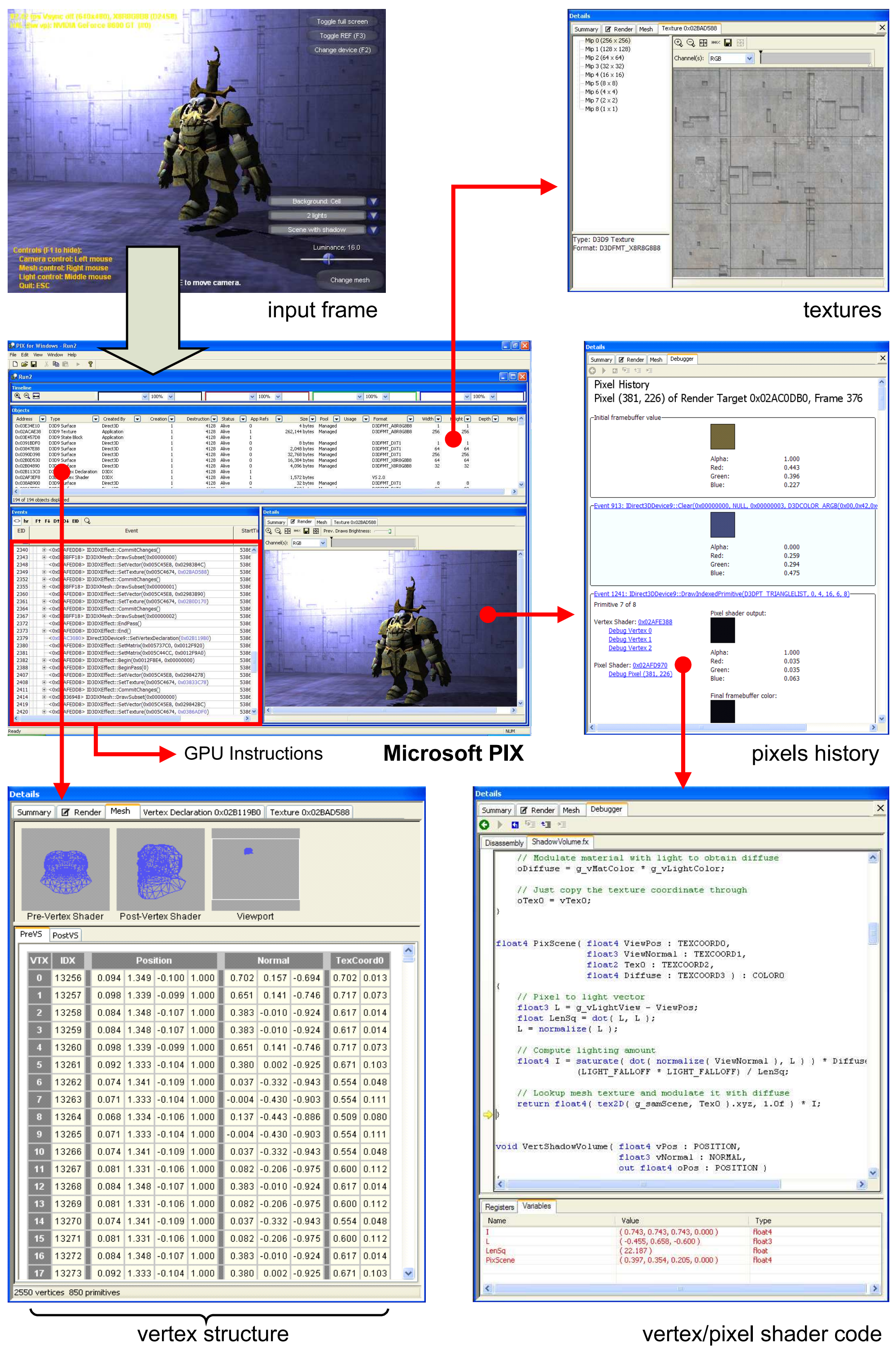
As mentioned earlier, a shader debugger hooks the graphics data stream sent by the application to the GPU to enable its analysis. To that end, the pipeline is emulated (typically via software) and the data flow interrupted on a frame-by-frame basis. Examples of data that can be captured are sequences of draw calls and registers used during the rendering process. These latter include frame buffers, depth buffers, texture buffers, vertex buffers and vertex and pixel shaders (Figure 7.3). These computer graphics concepts are explained in Chapter 2 and 4.

The purpose of redirecting such data to our testing system is not to change the final output of the game. Rather, we want to extend the original code of the virtual environment with new *testing* code in order to force the GPU to render the testing data needed for training and testing the detectors. In accordance with the previous chapters, we seek to generate *matrix maps*, retrieve *world-view-projection* matrices and synthesize images containing only shadows (see Chapters 4, 5 and 6).
A common sequence of DirectX rendering instructions that can be captured from the 3D API is shown in Algorithm 7.1. As can be observed, through the name of the calls it is easy to retrieve the information we need. In particular, the *shader code* in use can be identified through the function `SetTechnique` (line 2). The geometry to render can be detected via the function `SetVertexDeclaration`, by using the pointer to the vertices composing the object (line 3). The vertex structure (i.e. position, normal and tangent vectors) can be used to identify the target object without requiring the

[1]http://msdn.microsoft.com/en-us/directx/aa937788 (June 2011)

[2]http://www.deep-shadows.com/hax/3DRipperDX.htm

[3]http://glintercept.nutty.org/



Pixel History
Pixel (381, 226) of Render Target 0x02AC0DB0, Frame 376

Initial framebuffer value

Alpha:	1.000
Red:	0.443
Green:	0.396
Blue:	0.227

Event 913: IDirect3DDevice9::Clear(0x00000000, NULL, 0x00000003, D3DCOLOR_ARGB(0x00,0x42,0x

Alpha:	0.000
Red:	0.259
Green:	0.294
Blue:	0.475

Event 1241: IDirect3DDevice9::DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 4, 16, 6, 8)

Primitive 7 of 8

Vertex Shader: 0x02AFE388
Debug Vertex 0
Debug Vertex 1
Debug Vertex 2

Pixel Shader: 0x02AFD970
Debug Pixel (381, 226)

Pixel shader output:

Alpha:	1.000
Red:	0.035
Green:	0.035
Blue:	0.063

Final framebuffer color:

VTX	IDX	Position				Normal			TexCoord0	
0	13256	0.094	1.349	-0.100	1.000	0.702	0.157	-0.694	0.702	0.013
1	13257	0.098	1.339	-0.099	1.000	0.651	0.141	-0.746	0.717	0.073
2	13258	0.084	1.348	-0.107	1.000	0.383	-0.010	-0.924	0.617	0.014
3	13259	0.084	1.348	-0.107	1.000	0.383	-0.010	-0.924	0.617	0.014
4	13260	0.098	1.339	-0.099	1.000	0.651	0.141	-0.746	0.717	0.073
5	13261	0.092	1.333	-0.104	1.000	0.380	0.002	-0.925	0.671	0.103
6	13262	0.074	1.341	-0.109	1.000	0.037	-0.332	-0.943	0.554	0.048
7	13263	0.071	1.333	-0.104	1.000	-0.004	-0.430	-0.903	0.554	0.111
8	13264	0.068	1.334	-0.106	1.000	0.137	-0.443	-0.886	0.509	0.080
9	13265	0.071	1.333	-0.104	1.000	-0.004	-0.430	-0.903	0.554	0.111
10	13266	0.074	1.341	-0.109	1.000	0.037	-0.332	-0.943	0.554	0.048
11	13267	0.081	1.331	-0.106	1.000	0.082	-0.206	-0.975	0.600	0.112
12	13268	0.084	1.348	-0.107	1.000	0.383	-0.010	-0.924	0.617	0.014
13	13269	0.081	1.331	-0.106	1.000	0.082	-0.206	-0.975	0.600	0.112
14	13270	0.074	1.341	-0.109	1.000	0.037	-0.332	-0.943	0.554	0.048
15	13271	0.081	1.331	-0.106	1.000	0.082	-0.206	-0.975	0.600	0.112
16	13272	0.084	1.348	-0.107	1.000	0.383	-0.010	-0.924	0.617	0.014
17	13273	0.092	1.333	-0.104	1.000	0.380	0.002	-0.925	0.671	0.103

2550 vertices 850 primitives

```
    // Modulate material with light to obtain diffuse
    oDiffuse = g_vMatColor * g_vLightColor;

    // Just copy the texture coordinate through
    oTex0 = vTex0;
}


float4 PixScene( float4 ViewPos : TEXCOORD0,
                 float3 ViewNormal : TEXCOORD1,
                 float2 Tex0 : TEXCOORD2,
                 float4 Diffuse : TEXCOORD3 ) : COLOR0
{
    // Pixel to light vector
    float3 L = g_vLightView - ViewPos;
    float LenSq = dot( L, L );
    L = normalize( L );

    // Compute lighting amount
    float4 I = saturate( dot( normalize( ViewNormal ), L ) ) * Diffuse
               (LIGHT_FALLOFF * LIGHT_FALLOFF) / LenSq;

    // Lookup mesh texture and modulate it with diffuse
    return float4( tex2D( g_samScene, Tex0 ).xyz, 1.0f ) * I;
}


void VertShadowVolume( float4 vPos : POSITION,
                       float3 vNormal : NORMAL,
                       out float4 oPos : POSITION )
```

Name	Value	Type
I	(0.743, 0.743, 0.743, 0.000)	float4
L	(-0.455, 0.658, -0.600)	float3
LenSq	(22.187)	float
PixScene	(0.397, 0.354, 0.205, 0.000)	float4

Figure 7.3: Graphics data interception through Microsoft Pix$^{®}$ - Microsoft Pix is a shader debugger used for inspecting 3D application performances. Through a shader debugger, a large amount of data can be accessed in real-time. This include GPU instruction sequences, vertex and index buffers, shader codes, textures and pixel history (i.e. the history of processes through which pixels reach their final colors).

Algorithm 7.1: Common Rendering Instruction Sequence

```
1  SetRenderTarget(target_index, ordinary_render_target_pointer)
2  SetTechnique(ordinary_technique_handle)
3  SetVertexDeclaration(vertex_structure_pointer)
4  SetMatrix(matrix_name_handle, matrix_handle)
5  Begin(passes_pointer, flag)
6  BeginPass(0)
7    shader variables set up
8    CommitChanges()
9    DrawSubset(mesh_index)
10 ...
```

game to label it. Recall from Chapter 4 and 5 that object identification is needed in order to generate the matrix list, which links world-view-projection matrices and objects together. Finally, the world-view-projection matrix can be retrieved through the function `SetMatrix` and the name of the variable that it uses (line 4).

Once the interesting section of the graphics code has been identified, we will need to insert some additional instructions in order to generate the matrix maps. As explained in Chapter 4, different colors in the map will identify different matrices. Therefore, once a matrix has been extracted from the graphics code, it will need to be compared with the matrices that have been so far observed. This can be achieved via building hash tables of matrices and colors. If a matrix is novel, a new color will be generated and both matrix and color will be inserted in the table. If the input matrix is already in the table, then the related color will be read out and used to generate the matrix map. Finally, if all objects have been analyzed, the GPU will be instructed to render the map to a different target, rather than the ordinary frame buffer. This will prevent the matrix maps being overwritten by the frame buffer data. Algorithm 7.2 shows how this can be done.
Note that the entire set of calls used for rendering a single mesh is duplicated. The first line of the new set of calls (lines 1 to 9) sets the *testing technique* (see Algorithm 4.1) as the current shader effect to use. This technique is the one we used for drawing the matrix maps (Algorithm 4.1). Vertex structures and world-view-transformation matrices remain the same as the ones used in original code. Before rendering the matrix maps, a new variable needs to be set in the testing shader. This is the color of the current matrix (line 7), which is retrieved from the matrix table, as explained in Chapter 4. Finally, note how the matrix is rendered to a *test render target* (line 1) rather than the ordinary frame buffer (line 11). Linking matrix maps to the related

Algorithm 7.2: Adapted Rendering Instruction Sequence

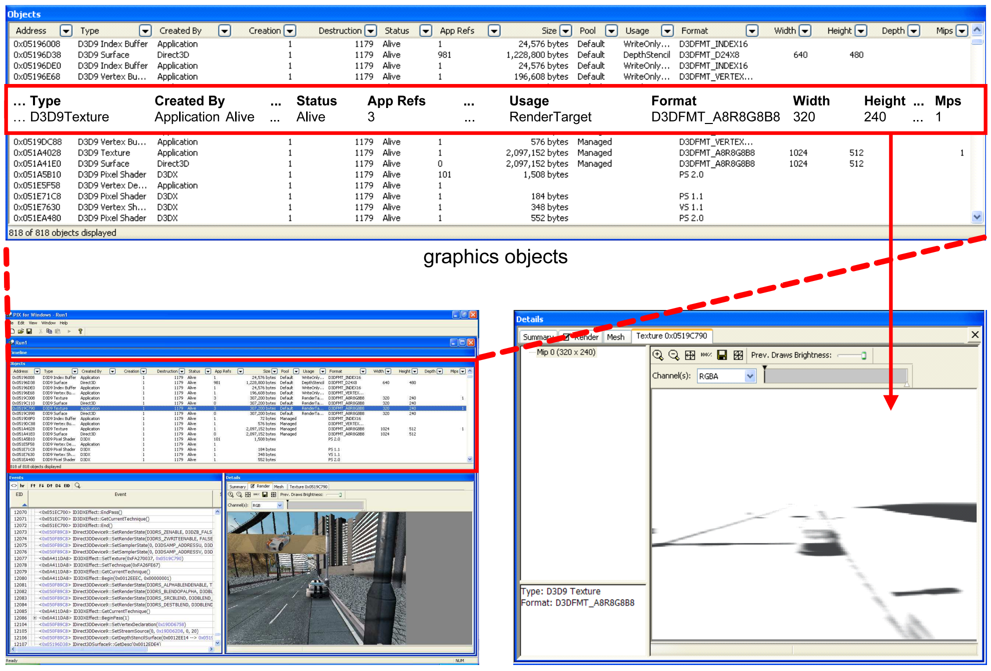
```
SetRenderTarget(target_index, test_render_target_pointer)
SetTechnique(testing_technique_handle)
SetVertexDeclaration(vertex_structure_pointer)
SetMatrix(matrix_name_handle, matrix_handle)
Begin(passes_pointer, flag)
BeginPass(0)
  matrix color set up
  CommitChanges()
  DrawSubset(mesh_index)
...
SetRenderTarget(target_index, ordinary_render_target_pointer)
SetTechnique(ordinary_technique_handle)
SetVertexDeclaration(vertex_structure_pointer)
SetMatrix(matrix_name_handle, matrix_handle)
Begin(passes_pointer, flag)
BeginPass(0)
  shader variables set up
  CommitChanges()
  DrawSubset(mesh_index)
...
```

world-view-projection matrix is an operation that does not need to be carried out by the graphics code. Rather, it will be performed by the detectors, as we explained in early chapters.

To complete the picture we will now show how to automate the appearance based mechanism for shadow aliasing detection. More precisely, we will elaborate on the way *shadow images* can be automatically extracted or synthesized through appropriate modifications to the graphics command stream. There are, principally, two ways in which rendering engines can add shadows to the final image. One approach uses a two-step process; first, a *shadow* gray-scale image in rendered in which the intensity of each pixel is proportional to the darkness of the shadow at that position; next, the pixel of the non-shadowed colored frame are darkened according to the shadow image. In the second approach, shadowing and coloring are performed in one step, after determining the amount of darkness the pixel is expected to exhibit. We will discuss both solutions in the following.

If the shadow image is generated by the game directly, all we need to do is intercept the data stream to retrieve that data. This is done by following the approach used for generating the matrix maps. In this case, the challenge is to identify the *shadow image* texture among the other textures used by the game. To that end, we note that the shadow image texture presents distinctive features. On the one hand, such an image is typically rendered as a low resolution texture (typically 8 bits), of the same size as the frame buffer and it is composed by only one mipmap level (Figure 7.4). Also, the texture *set up* operation takes place in proximity of alpha blending and depth buffer operations. In order to facilitate the identification of shadow maps in the graphics code, the original shader code could as well be augmented with special *tags* or *descriptors* that will inform the detector about the type of resource analyzed.

Once the shadow texture is identified, it can be analyzed with the technique introduced in the previous chapter.



Objects

Address	Type	Created By	Creation	Destruction	Status	App Refs	Size	Pool	Usage	Format	Width	Height	Depth	Mips
0x05196008	D3D9 Index Buffer	Application	1	1179	Alive	1	24,576 bytes	Default	WriteOnly...	D3DFMT_INDEX16				
0x05196D38	D3D9 Surface	Direct3D	1	1179	Alive	981	1,228,800 bytes	Default	DepthStencil	D3DFMT_D24X8	640	480		
0x05196DE0	D3D9 Index Buffer	Application	1	1179	Alive	1	24,576 bytes	Default	WriteOnly...	D3DFMT_INDEX16				
0x05196E68	D3D9 Vertex Bu...	Application	1	1179	Alive	1	196,608 bytes	Default	WriteOnly...	D3DFMT_VERTEX...				
0x0519DC88	D3D9 Vertex Bu...	Application	1	1179	Alive	1	576 bytes	Managed		D3DFMT_VERTEX...				
0x051A4028	D3D9 Texture	Application	1	1179	Alive	1	2,097,152 bytes	Managed		D3DFMT_A8R8G8B8	1024	512		1
0x051A41E0	D3D9 Surface	Direct3D	1	1179	Alive	0	2,097,152 bytes	Managed		D3DFMT_A8R8G8B8	1024	512		
0x051A5B10	D3D9 Pixel Shader	D3DX	1	1179	Alive	101	1,508 bytes			PS 2.0				
0x051E5F58	D3D9 Vertex De...	Application	1	1179	Alive	1								
0x051E71C8	D3D9 Pixel Shader	D3DX	1	1179	Alive	1	184 bytes			PS 1.1				
0x051E7630	D3D9 Vertex Sh...	D3DX	1	1179	Alive	1	348 bytes			VS 1.1				
0x051EA480	D3D9 Pixel Shader	D3DX	1	1179	Alive	1	552 bytes			PS 2.0				

818 of 818 objects displayed

... Type	Created By	...	Status	App Refs	...	Usage	Format	Width	Height	...	Mps
... D3D9Texture	Application Alive	...	Alive	3	...	RenderTarget	D3DFMT_A8R8G8B8	320	240	...	1

Type: D3D9 Texture
Format: D3DFMT_A8R8G8B8

Figure 7.4: Example of Shadow Texture Extraction - Shadows in virtual environments are often cast through *shadow textures* (b). These are images produces during the rendering process that will eventually be blended with the final frame (a). Like any other graphics data, shadow textures can be extracted through graphics data interception, and identified through distinctive graphics properties, such as resolution, number of mipmap levels and usage (top image).

If the shadow image cannot be extracted from the pipeline, a slight modification to the shader code needs to be introduced, as shown in Algorithm 7.3.
The highlighted code (line 13 to 21) shows the instructions that need to be added to the original rendering code in order to generate an image with no shadows in it. As it can be observed, the shadow map texture (addressed by `shadow_map_name_handle`)

Algorithm 7.3: Adapted Shadow Mapping Instruction Sequence

```
1  SetRenderTarget(target_index, ordinary_render_target_pointer)
2  SetTechnique(testing_technique_handle)
3  SetVertexDeclaration(vertex_structure_pointer)
4  SetMatrix(matrix_name_handle, matrix_handle)
5  Begin(passes_pointer, flag)
6  BeginPass(0)
7     shader variables set up
8     SetTexture(shadow_map_name_handle, texture_handle)
9     CommitChanges()
10    DrawSubset(mesh_index)
11 ...
12 SetRenderTarget(target_index, test_render_target_pointer)
13 SetTechnique(testing_technique_handle)
14 SetVertexDeclaration(vertex_structure_pointer)
15 SetMatrix(matrix_name_handle, matrix_handle)
16 Begin(passes_pointer, flag)
17 BeginPass(0)
18    shader variables set up
19    SetTexture(shadow_map_name_handle, texture_handle)
20    CommitChanges()
21    DrawSubset(mesh_index)
22 ...
```

will need to be computed through a modified version of the shadow map algorithm (line 13) and rendered to a texture (line 12), rather than to screen (line 1). The rest of the testing code is a copy of the standard code. The "test version" of the shadow map algorithm consists in generating a flat shadow map whose value is equal to the maximum depth a pixel can have (i.e. 1). In this way, all pixels seen by the camera will have a related equal or smaller distance from the light source, and hence they will not be considered in shadow by the shadow map algorithm. Having the ordinary shaded image, and a new image containing no shadows, enables the rendering of a frame with shadows only. This is done by performing a pixel-wise XNOR operation between the shaded image and the image with no shadows. The result of the XNOR operation is depicted in Figure 7.5.

7.3 Summary

In this chapter we have shown how training and testing data can be automatically extracted from virtual environments. Human-like stimuli can be generated by monitoring the dynamics of the avatar in the virtual world. The simplest way to achieve this is via

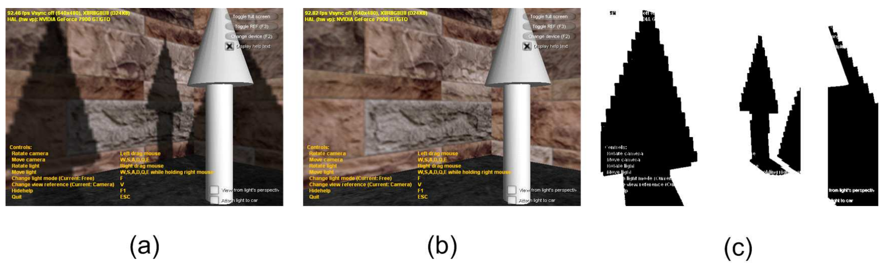


Figure 7.5: Extraction of shadows from an image - A XNOR operation performed between the original image (a) and a non-shadowed version of it (b) produces a B&W image containing only shadows (c).

linking the world matrix of an avatar to reference points of the world, through a lookup table. However, more sophisticated mechanisms can be devised as additional environment information becomes available, perhaps through standard software interfaces. The graphics data needed for training and testing the classifiers can be automatically extracted from the rendering system. We have discussed how this can be done via intercepting the data stream at the driver level. To that end, the target graphics instruction sequences need first to be identified. Then, additional testing code can be inserted into the stream. Finally, the modified GPU code can be sent to the graphics hardware, which will produce the ordinary output as well as the required testing data. This chapter shows a theoretical analysis of the feasibility of such an approach, its implementation is left to future work. In particular, further research can be oriented to explore ways of identifying the sections of the graphics stream that need to be modified, given the testing activity to perform. One way to achieve this, is to augment games software with appropriate metadata. Similar research has been conducted for compiling natural semantics into highly efficient code [111] and for semantic error classification in large software systems [37].

8

Discussion and Conclusion

With this research work we set out to quantify and characterize unwanted visual artifacts that may emerge during player-virtual environment interaction. We commenced by providing a formalization of our research problem and framed our research within the context of detection of inconsistent outputs generated by the rendering function of a 3D application. An investigation into the typology of visual artifacts that may appear during a play-session enabled us to classify all virtual environment bugs into three major categories, viz., *Entertainment Issues*, *Usability Issues* and *Environment Inconsistencies* (Chapter 2). Entertainment Issues concern bugs in the game AI, mechanics, play and balance. Game usability issues arise when the game becomes unplayable, hard to play or prevents the game mechanics unfolding normally. Finally, environment inconsistencies relate to the environment consistency, which is the ability of the environment to maintain the level of realism meant to be offered. This study focused on the detection of a sub-set of environment inconsistencies, these are: geometry corruption, color and shadowing malfunctions. Although environment inconsistencies are issues related to computer graphics technology, their formalization and automatic detection required us to explore findings from disciplines such as machine learning and computer vision. This is not surprising as the aim of building visual consistency detectors is, in effect, seeking to imitate some human visual anomaly detection capabilities. In this chapter, we summarize our findings through the research questions we posed in Chapter 1, that we are now ready to answer:

Can visual anomalies in virtual environments be defined in an environment-independent manner?

We have shown that some geometry and color issues can be effectively defined and detected in an environment-independent manner (Chapters 4 and 5). Although both anomalous and correct visualizations may vary a great deal throughout a play-session, we have shown how the canonical object space representation can be used to model the correct behaviour of the rendering system. We have also shown how good models of rendering behaviour can effectively execute the anomaly detection task.

We noted that not all visual inconsistencies are environment-independent, however. Global illumination malfunctions are examples of environment-dependent issues. In such cases, the normal behaviour of the rendering system can only be defined in a context-dependent fashion, that is to say, via accessing some contextual information, internal to the game. Accessing large amounts of internal graphics data has two significant disadvantages: it may interfere with the normal behaviour of the system to test; and it may require the extraordinary effort of designing testing interfaces, from the game designer. A way to overcome this difficulty is to endow the detectors with *a priori* knowledge of the anomalous artifact. The detectors should be designed in such a way as to look for physical properties peculiar to the anomaly, in the frame at hand. In Chapter 6, we have shown how this can be achieved for one specific anomaly affecting a shadowing algorithm, via an appearance based scheme. Performing appearance based detection may provide simple, effective and robust alternatives to building reference software for debugging purposes.

What mechanisms and descriptors are good to effectively discern between anomalous and valid visualizations?

Both model based and view based approaches turned out to be good approaches for detecting context-independent bugs. From our experiments, the model based technique proved to be more suitable for detecting geometry issues. By contrast, the view based classifiers were able to learn complex color changes.

For the model based classifier (Chapter 4), the *Hausdorff* metric proved to be a very good measure of geometric consistency. As far as colors are concerned, we have observed that the *Bhattacharyya* distance performed best in almost all test cases. The performance of the view based classifiers is largely decided by the descriptors (feature vectors) used. In general, the goodness of a descriptor depends on the type of anomaly to detect, the color and geometric complexity of the target object geometry and the model used for capturing the target behaviour of the rendering mechanism. Good geometry features are the area and eccentricity of the 2D object silhouette. The mean distance between camera and object was found to be a sub-optimal feature choice, with regards to mesh corruption bugs. The goodness of color features for the model based approach is decided by the type of environment considered. Cartoon-like environments seem to be best described by *Hue* histograms only. As the level of realism increases, including *Saturation* and *Value* components into the color descriptors seems to improve the performance of the detectors. Both ANN and SOM model based mechanisms proved to be good at discriminating between correct and anomalous geometry issues. The ANN detectors, however, seemed to be unsuitable for modelling color changes. Depending on the type (ANN versus SOM) and complexity (number of neurons) of the network used, we found that the accuracy may be further improved by selecting an appropriate cluster radius[1]. We noticed that the size of the cluster radius may influence both geometry and color detection.

We have not introduced general approaches to the detection of context-dependent anomalies. However, we note that some global effect issues can be addressed through the view based scheme. We have shown evidence of this for specular illumination artifacts (Chapter 5, Section 5.4.4); we postulate that similar results can be obtained for simple reflection effects (e.g. *environment mapping*). For more complex global effects, we argued that the appearance based scheme should be considered instead.

Will such descriptors have an object space or an image space representation?

If we follow the appearance based scheme, anomalies are inherently represented in image space. If we consider the shadow aliasing example (Chapter 6), we find that the anomaly (i.e. jagged shadow edges) is described through the second moment matrix, which is defined in image space. By contrast, the view based schemes handles descriptors lying on "hybrid" manifolds. For example, the ANN view based approach, accepts the world-view-projection matrix (the object-to-screen space transformation matrix) as an input and predicts geometry or colors on screen. The SOM detectors concatenate the transformation matrices with 2D shape and color information. Finally, in the model based scheme (Chapter 4) both geometries and colors are defined and measured in object space. In general, the space in which events (consistent or otherwise) are

[1]We referred to the cluster radius as the size of the sub-parts into which the original geometry will be segmented, prior to training and testing

described depends on the type of anomaly to detect.

How much can we treat the engine as a black box in the process of finding visual errors?

Under the black-box assumption, any anomaly detection mechanisms should make inference upon only the input to, and output from, the virtual environment to test. Building an ideal inference machine which works under the black-box assumption equals imitating human bug-detection capabilities. In practice, robust solutions based on the available technology will necessarily need to rely on some internal graphics data from the game. In such cases, it is important that the exchange of extra information between virtual environment, and system to test, does not undermine the functioning of the environment itself. In Chapter 7 we have suggested ways in which the testing data can be extracted from the game in an application-independent manner, with minimal interference to the virtual environment software.

8.1 Limitations

The approaches introduced in this work assess appearance on the basis of the information concerning the target object only, while ignoring the rest of the scene. We have shown that contextual information is not needed if context-independent anomalies are to be detected. In fact, even some environment interference — such as occlusions, clutter and simple global illumination effects — can be handled effectively through object segmentation and a view based scheme, as we have shown in Chapter 4 and 5.

In this work we have not provided a general solution to the automatic detection of context-dependent visual artifacts, such as collision detection issues, realistic reflection, radiosity and refraction effects — amongst others. Also, the consistency we have defined is assessed at the "local" scale, that is, at the level of individual objects or object components rather than the visual relationship between them. It is important to point out that even though, at a low level, the information from all parts of an object may appear consistent with the user expectation, at a higher level, the parts may appear in an odd mutual relationship. As an example, imagine a car whose wheels, glasses and body are properly visualized but in the wrong position or size. Finally, note that even if an object in its wholeness may appear consistent, its relation with the rest of the scene may not be in such a state. Similar considerations apply to the consistent appearance of an object, in its wholeness, with respect to the rest of the scene.

The assumption under which our geometry and color anomaly detectors operate is that the target geometry is non-deformable. All vertices, and thus pixels, of the target object are rendered through the same world-view-projection matrix, for a given frame.

8.2 Computational Complexity of the Detectors

An analysis of the asymptotic complexity of our algorithms was not performed. Training and testing time of model based and view based approaches mostly depends on the size (screen area in pixels) of the objects or events to analyze; and on the complexity of the models used, that is, cluster radii for the colored point clouds and number of hidden or output neurons for the ANNs and SOMs respectively. The efficiency of the appearance based classifier depends on the array of integration scales used and the resolution (in pixels) of the frames to test.

All algorithms presented in this work have been implemented in Matlab® and tested on a Pentium 4, 3.00GHz, 3GB of RAM. Being an interpreted language, Matlab exhibits execution times that are often smaller, compared to other compiled languages. On an average, the best performing model based and view based classifiers performed detection at the rate of 5 seconds per frame, for 640×480 images. The data collection and training time was of about 20 minutes per object, across a collection of about 200 training images. Data collection consisted of retrieving the data (pixels position and color) of the objects of interest for each validated frame; training consisted of building the colored point cloud or training the neural networks. As far as concerns the appearance based detector, no training was required and a single 640×480 image was processed in about 30 seconds.

8.3 Future Research

The results of this study open up a number of avenues for future research. As far as the view based detectors are concerned, it would be interesting to see how performance varies when using other geometry descriptors that have not been considered in this work. Such alternative descriptors could for example be the affine invariant Fourier descriptor (AIFD), the CSS descriptor or the generalized Hough transform descriptor (GHTD). In Chapter 5, we postulated that these descriptors were less suitable to serve our purposes; mainly due to their *boundary based* nature (i.e. they only rely on the information about the contour of the object). Such an hypothesis could be easily tested by re-training and testing the view based classifiers with the aforementioned descriptors.

Further research could be undertaken to explore mechanisms of automatic data extraction from the applications to test. Although the data used by our algorithms does not require extensive software modifications, we did customize the original applications in order to generate the training and testing data. In Chapter 7 we have discussed possible solutions to the automatic extraction of testing data. Implementing those ideas will help in reducing the influence from the testing device to the 3D application to test.

Fast implementations of our detectors could also be the subject of future research. Our techniques, in their current implementation, do not allow real time debugging, as the testing time is in the order of dozens of seconds, let alone the training time, which is in the order of minutes. To significantly speed up both training and recognition, all per-pixel-wise operations (e.g. the inverse-transformation of coordinates and colors from screen to object space) and patch-wise filtering (e.g. convolution) could be run by graphics software and hardware, exploiting the high parallelism of modern GPUs.

All geometry and color anomaly detectors, in their current state, require the target meshes to be non-deformable. Deformable meshes are those whose final shape on screen is decided by multiple transformation matrices, typically, one matrix per vertex, as opposed to one matrix per mesh in the case of non-deformable meshes. Further research can be conducted in order to relax this assumption. Recall from Chapters 4 and 5 that the transformation matrices where stored through matrix maps and tables, in order to determine where the objects where on screen and to learn the object appearance. Because a deformable mesh undergoes vertex-wise geometric operations, allowing for deformable meshes will probably require the matrix maps and tables to identify vertex regions, rather than object parts. Object segmentation (i.e. labelling which pixel belongs to which object) can therefore be accomplished by metric space decomposition (e.g. Voronoi partitioning [10]) given the vertex position on screen.

Our algorithm can deal with a large family of context-independent anomalies. In order to provide general solutions to the detection of context-dependent bugs, one could explore ways in which the surrounding environment can be described. We believe that useful descriptions of context-dependent appearances will need to rely on some *a priori* information about the environment. Clearly, the less data required from the game, the more feasible and appealing the solution. As for this research, the challenge will lie in discovering *one fits all* feature spaces and models, that is, descriptors and detectors that can be plugged into any virtual environment in order to uncover a comprehensive enough family of bugs — context-dependent or otherwise.

From a psychophysics standpoint, it would be interesting to determine to which extent the approaches proposed in this research perform better than human players.

In order to enable this comparison, the testing system would first need to be extended so as to avoid reporting the same bug multiple times. As an example, consider the case in which the geometry of the avatar controlled by the player, becomes corrupted over a set of frames. A human player will have no difficulty determining that the bug is the same, for the anomaly is consistent across different images and it concerns the same object or object part. Thus, she will report the anomaly only once. By contrast, our algorithms perform detection on a frame-to-frame basis, considering the same object, over different frames, as different entities. By giving the anomalies an object or context-dependent *signature*, the system can be enabled to produce human-like reports, and thus, the performance human versus machine can be measured. Along this line, computational models of the visual human system could also be explored in order to model errors from a perceptual point of view. Perceptual error metrics have been successfully applied in computer graphics to accelerate rendering [152]. Research in this direction could allow determining what types of visual anomalies are likely to be detected by a human observer.

Finally, the list of detectable bugs could be extended so as to include other environment malfunctions whereby the interaction between virtual objects is affected. Examples are inconsistent or meaningless actions performed by non-playing characters and unrealistic physics behaviours. These types of malfunctions are context-dependent, for the meaning of the interaction depends on the context in which the interaction unfolds. We suggest that this type of environment consistency can be assessed through techniques similar to the ones proposed in this research. A likely solution may come by using the same transformation matrices used by the anomaly detectors we have proposed. Differently from our consistency detectors, however, we suggest that in order to debug a virtual interaction, one should only look at the time trajectories of the internal game data (i.e. the geometric transformation matrices), and neglect the way such data is rendered to screen.

8.4 Summary

The results of this thesis provide supporting evidence for our hypothesis that visual consistency can be quantitatively defined and automatically measured in virtual environments, for a large family of visual inconsistencies. We have provided general measures of consistency based on different representations of the target object and/or anomaly. Our major findings suggest that the object space information from the game can be effectively combined with the screen space description of the virtual object. In particular, the 3D object space representation of objects can be reconstructed from the frames produced by the game in order to effectively access geometry. Also, we found that the transformation matrices can be used to learn the color changes that objects undergo during a play session. Feature vectors combining transformation matrices and color or geometry descriptors form hyperdimensional manifolds of consistent visualizations. Learning the topology of such manifolds (e.g. through Self Organizing Maps) enables the effective detection of anomalous or inconsistent appearances.

Fast and accurate anomaly detectors will ease the burden of beta testers and developers of 3D applications. On the one hand, the testing system can be designed in such a way as to report the anomalies to the designer, as a beta tester would normally do. On the other hand, statistical analyses can be performed on the large amounts of data produced by the detectors in order to assist the identification of the root cause of the artifacts. Pattern recognition techniques applied to the test data may, for example, reveal that some geometries are affected by mesh corruption, with statistically significant higher-than-average frequency. Other results may show that texture issues affect some environment regions or geometries more frequently than others. As one can imagine, the test results can be analyzed in a number of different ways. As is always the case, the type of analysis performed depends on the phenomenon that one seeks to observe.

References

[1] Fuga - the fun of gaming: Measuring the human experience of media enjoyment. http://project.hkkk.fi/fuga/, June 2011. 4

[2] E. Alhoniemi, J. Hollmn, O. Simula, and J. Vesanto. Process monitoring and modeling using the self-organizing map. *Integrated Computer Aided Engineering*, 6:3–14, 1999. 101

[3] S. Ando. Clustering needles in a haystack: An information theoretic analysis of minority and outlier detection. In *IEEE International Conference on Data Mining*, pages 13–22, 2007. 43

[4] M.V. Aponte, G. Levieux, and S. Natkin. Scaling the level of difficulty in single player video games. In *Proceedings of the 8th International Conference on Entertainment Computing*, ICEC '09, pages 24–35, Berlin, Heidelberg, 2009. Springer-Verlag. 13

[5] ESA Entrtainment Software Association. Industry facts. World Wide Web electronic publication (http://www.theesa.com/), 2009. 1

[6] F. Bacao, O Bação, V. Lobo, and M. Painho. Self-organizing maps as substitutes for k-means clustering. In *International Conference on Computational Science*, pages 476–483. Springer-Verlag, 2005. 103

[7] B. Balamuralithara and P. C. Woods. Virtual laboratories in engineering education: The simulation lab and remote lab. *Computer Applications in Engineering Education*, 17(1):108–118, 2009. 2

[8] D. H. Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111–122, 1981. 92

[9] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *In ECCV*, pages 404–417, 2006. 46

[10] M. Berg. *Computational geometry: algorithms and applications*. Springer, Berlin, 2nd edition, 2000. 158

[11] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, USA, 1996. 37, 99

[12] J. F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12:286–292, August 1978. 16, 20

[13] R. Bogacz, M. W. Brown, and C. Giraud-Carrier. High capacity neural networks for familiarity discrimination. In *In Proceedings of ICANN99*, pages 773–778, 1999. 37, 38

[14] M. Botsch and L. Kobbelt. High-quality point-based rendering on modern gpus. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 335, Washington, DC, USA, 2003. IEEE Computer Society. 18

[15] H. A. Bourlard and N. Morgan. *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. 98

[16] E. Boylan. Equiconvergence of martingales. *Ann. Math. Statist.*, 42:552–559, 1971. 58

[17] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984. 76, 79

[18] C. Browne. *Automatic generation and evaluation of recombination games*. eprints.qut.edu.au, 2010. 6

[19] C. Campbell and K. P. Bennett. A linear programming approach to novelty detection. In *NIPS*, pages 395–401, 2000. 39

[20] J. F. Canny. *A computational approach to edge detection*, pages 184–203. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987. 131

[21] E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Department of Compuer Science, University of Utah, 1974. 15

[22] F. Chaker, M. T. Bannour, and F. Ghorbel. Contour retrieval and matching by affine invariant fourier descriptors. In *MVA'07*, pages 291–294, 2007. 92

[23] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):1–58, 2009. 33, 34, 35, 41, 42, 43, 90

[24] Varum Chandola. *Anomaly Detection for Symbolic Sequences and Time Series Data*. PhD thesis, University of Minnesota, 2009. 34

[25] C. Chen, Y. Hung, and J. Cheng. Ransac-based darces: A new approach to fast automatic registration of partially overlapping range images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21:1229–1234, 1999. 44

[26] F. S. Cohen and J. Wang. Part ii: 3-d object recognition and shape estimation from image contours using b-splines, shape invariant matching, and neural network. *IEEE Transactions in Pattern Analysis and Machine Intelligence*, 16(1):13–23, 1994. 44

[27] E. Corchado and Á. Herrero. Neural visualization of network traffic data for intrusion detection. *Appl. Soft Comput.*, 11:2042–2056, March 2011. 33

[28] B.C. Csaji. *Approximation with Artifcial Neural Networks*. PhD thesis, Eotvos Lorand University, 2001. 99

[29] K. Das and J. G. Schneider. Detecting anomalous records in categorical datasets. In *Knowledge Discovery and Data Mining*, pages 220–229. ACM Press, 2007. 36

[30] M. David and E. Touradj. Efficient rotation-discriminative template matching. In *Iberoamerican Congress on Pattern Recognition (CIARP)*, Lecture Notes in Computer Science (LNCS), pages 221–230, Valparaiso, Chile, 2007. Springer-Verlag. 62

[31] H. Demuth and M. Beale. *Neural Network Toolbox: For use with MATLAB: User's Guide*. The Mathworks, 1993. 99, 104

[32] J. Denzinger. Exploratory testing for unwanted behavior using evolutionary learning techniques. Tech. report, University of Calgary, Alberta, Canada, 2007. 6, 7, 13

[33] H. Desurvire, M. Caplan, and J. A. Toth. Using heuristics to evaluate the playability of games. In *CHI '04: CHI '04 extended abstracts on Human factors in computing systems*, pages 1509–1512, New York, NY, USA, 2004. ACM. 12

[34] M. Dickheiser. *Game Programming Gems 6 (Book & CD-ROM) (Game Development Series)*. Charles River Media, Inc., Rockland, MA, USA, 2006. 2

[35] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2 edition, Nov 2001. 42

[36] V. Emamian, M. Kaveh, A. H. Tewfik, Z.i Sh, L. J. Jacobs, and J. Jarzynski. Robust clustering of acoustic emission signals using neural networks and signal subspace projections. *EURASIP Journal on Advances in Signal Processing*, 2003(3):276–286, 2003. 38

[37] N. J. G. Falkner. Ontologically-based context checking in arbitrary programming languages. In *Proceedings of the 19th International Conference on Database and Expert Systems Application*, pages 220–224, Washington, DC, USA, 2008. IEEE Computer Society. 152

[38] D. M. Farid, N. Harbi, S. Ahmmed, M. Z. Rahman, and C. M. Rahman. Mining network data for intrusion detection through nave bayesian with clustering. In *International Conference on Computer, Electrical, System Science, and Engineering (ICCESSE'10)*, Paris, France, June 2010. 36

[39] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 47

[40] A. Fink, J. Denzinger, and J. Aycock. Extracting npc behavior from computer games using computer vision and machine learning techniques. In *IEEE Symposium on Computational Intelligence and Games*, pages 24–31, Hawaii, 2007. IEEE Press. 6

[41] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, March 1974. 69

[42] J. Fodor and Z. Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, (28):3–71, 1988. 98

[43] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice in C*. Addison-Wesley Professional, second edition, August 1995. 17, 18

[44] S. Fredrickson, S. Roberts, N. Townsend, and L. Tarassenko. Speaker identification using networks of radial-basis functions. In *Seventh European Signal Processing Conference, EUSIPCO'94*, Edinburgh, Scotland, 1994. 37

[45] R. Fujimaki. An approach to spacecraft anomaly detection problem using kernel feature space. In *Ninth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*. ACM Press, 2005. 33

[46] A. B. Gardner, A. M. Krieger, G. Vachtsevanos, and B. Litt. One-class novelty detection for seizure analysis from intracranial eeg. *J. Mach. Learn. Res.*, 7:1025–1044, December 2006. 33

[47] C. L. Giles, S. Lawrence, and A. C. Tsoi. Noisy time series prediction using a recurrent neural network and grammatical inference. In *Machine Learning*, pages 161–183, 2001. 98

[48] A. Golovinskiy, V. G. Kim, and t. Funkhouser. Shape-based recognition of 3d point clouds in urban environments. In *International Conference on Computer Vision (ICCV)*, September 2009. 44

[49] A.I. Gonzalez, M. Graña, A. D'Anjou, F.X. Albizuri, and M. Cottrell. A sensitivity analysis of the self organizing maps as an adaptive one-pass non-stationary clustering algorithm: the case of color quantization of image sequences. *Neural Processing Letters*, 6(3):77–89, 1997. 103

[50] K. Graft. Stardock reveals impulse, steam market share estimates. World Wide Web electronic publication, November 2009. 11

[51] M. Gross, R.W. Sumner, and N. Thürey. The design and development of computer games. *The Design of Material, Organism, and Minds*, 2:39–51, 2010. 2

[52] S. Guttormsson, R. J. Marks II, M. A. El-Sharkawi, and I. Kerszenbaum. Elliptical novelty grouping for on-line short-turn detection of excited running rotors. *IEEE Transaction on Energy Conversion*, 14(1):16–22, 1999. 40

[53] C. Harris and M. Stephens. A combined corner and edge detector. In *ALVEY Vision Conference*, pages 147–151, Cambridge, UK, 1988. 132

[54] M. Harris. Mapping computational concepts to gpus. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 50, New York, NY, USA, 2005. ACM. 49

[55] T. Harris. Neural network in machine health monitoring. Professional Eng., July/August 1993. 38

[56] J.-M. Hasenfratz, M. Lapierre, N. Holzschuch, and F.X. Sillion. A survey of real-time soft shadows algorithms, 2003. 16, 127

[57] S. Hawkins, H. He, G. Williams, and R. Baxter. Outlier detection using replicator neural networks. In *Fifth Int. Conf. and Data Warehousing and Knowledge Discovery (DaWaK02)*, pages 170–180, 2002. 37

[58] D. D. Hearn and M. P. Baker. *Computer Graphics with OpenGL*. Prentice Hall Professional Technical Reference, 2003. 18

[59] D.O. Hebb. *The organization of behavior: a neuropsychological theory*. L. Erlbaum Associates, 2002. 38

[60] K. Hempstalk and E. Frank. Discriminating against new classes: One-class versus multi-class classification. In *Australian Joint Conference on Artificial Intelligence*, 2008. 36

[61] T. V. Ho and J. Rouat. Novelty detection based on relaxation time of a network of integrate-and-fire neurons. In *2nd IEEE World Congress on Computational Intelligence*, 1998. 39

[62] R.V. Hogg and J. Ledolter. *Engineering statistics*. Mathematics & statistics. Macmillan, 1987. 79, 80, 83

[63] A. Hyvärinen, J. Karhunen, and E.i Oja. *Independent Component Analysis*. Wiley-Interscience, 1 edition, May 2001. 45

[64] Tarnanas I. and Adam D. Sonic intelligence as a virtual therapeutic environment. *CyberPsychology and Behavior*, 6(3):309–314, Jun 2003. 2

[65] D. Irish. *The Game Producer's Handbook*. Course Technology Press, Boston, MA, United States, 2005. 3, 5

[66] C. Jacquemin, B. Planes, and R. Ajaj. Shadow casting for soft and engaging immersion in augmented virtuality artworks. In *ACM Multimedia*, pages 477–480, 2007. 127

[67] A. Jagota. Novelty detection on a very large number of memories stored in a hopfield-style network. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, page 905, Seattle, WA, 1991. 37

[68] M. Johnson, A. andl Hebert. Using spin images for efficient object recognition in cluttered 3d scenes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(1):433–449, May 1999. 44

[69] I. T. Jolliffe. *Principal Component Analysis*. Springer, second edition, October 2002. 45

[70] T. Joshi, B. Vijayakumar, D.J. Kriegman, and J. Ponce. Hot curves for modeling and recognition of smooth curved 3d objects. *IVC*, 15(7):479–498, July 1997. 44

[71] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J. P. Laulajainen, R. Carmichael, V. Poulopoulos, A. Laikari, P. Perälä, A. De Gloria, and C. Bouras. Platform for distributed 3d gaming. *International Journal of Computer Games Technology*, 2009:1–15, January 2009. 145

[72] S. Kaski. Data exploration using self-organizing maps. *Acta Polytechnica Scandinavica, Mathematics, Computing and Management in Engineering*, (82), 1997. 101

[73] Y. Ke and R. Sukthankar. Pca-sift: a more distinctive representation for local image descriptors. In *2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2, pages 506–513, 2004. 46

[74] F. Keinosuke. *Introduction to statistical pattern recognition (2nd ed.)*. Academic Press Professional, Inc., San Diego, CA, USA, 1990. 62

[75] O. Khayat, H. R. Shahdoosti, and A. J. Motlagh. An overview on model-based approaches in face recognition. In *AIKED'08: Proceedings of the 7th WSEAS International Conference on Artificial intelligence, knowledge engineering and data bases*, pages 109–115, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS). 44

[76] E. M. Knorr, R. T. Ng, and V. Tucakov. Distance-based outliers: Algorithms and applications. *The Vldb Journal*, 8:237–253, 2000. 40

[77] T. Kohonen. *Self-organization and associative memory*. Springer-Verlag New York, Inc., New York, NY, USA, 1989. 38, 100

[78] L. Kotoulas and I. Andreadis. Colour histogram content-based image retrieval and hardware implementation. *Circuits, Devices and Systems*, 150(5):387–393, 2003. 62

[79] S. Kozlov. *GPU Gems - Perspective Shadow Maps: Care and Feeding*, chapter 14, pages 217–244. Pearson Higher Education, 2004. 129

[80] S. Laine, H. Saransaari, J. Kontkanen, J. Lehtinen, and T. Aila. Incremental instant radiosity for real-time indirect illumination. In *Eurographics Symposium on Rendering*, pages 277–286. Citeseer, 2007. 20

[81] J. E. Laird. An exploration into computer games and computer generated forces. In *9th Conference on Computer Generated Forces and Behavioural Representations*, pages 241–250, Orlando, FL, May 2000. 2

[82] A. Lauritzen. *GPU Gems 3 - Summed-Area Variance Shadow Map*, chapter 8, pages 157–182. Addison-Wesley Professional, 2005. 49

[83] D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999. 45

[84] K. Leung and C. Leckie. Unsupervised anomaly detection in network intrusion detection using clusters. In *28th Australasian Computer Society Conference*, volume 38, pages 333–342, 2005. 35

[85] Y. Li, M. J. Pont, and N. B. Jones. Improving the performance of radial basis function classifiers in condition monitoring and fault diagnosis applications where 'unknown' faults may occur. *Pattern Recognition Letters*, 23(5):569– 577, 2002. 37

[86] W. Liang and M. C. C. Tan. *Vision and virtuality: The construction of narrative space in film and computer games*, chapter 5, pages 98–109. Wallflower Press, London, 2002. 4

[87] G. Liktor and C. Dachsbacher. Real-time volumetric caustics with projected light beams. In *Proceedings of 5th Hungarian Conference on Computer Graphics and Geometry*, 2010. 20

[88] C. Lindley and L. Nacke. Boredom, immersion, flow - a pilot study investigating player experience. In *IADIS Gaming 2008: Design for Engaging Experience and Social Interaction,*. IADIS, 2008. 3

[89] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004. 44, 46

[90] F. Luna. *Introduction to 3D Game Programming with Direct X 9.0c: A Shader Approach (Wordware Game and Graphics Library)*. Wordware Publishing Inc., Plano, TX, USA, 2006. 20, 21, 49, 64, 65

[91] J. Ma and S. Perkins. Time-series novelty detection using one-class support vector machines. *Proceedings of the International Joint Conference on Neural Networks 2003*, 3:1741–1745, 2003. 39

[92] D. J. C. MacKay. Bayesian interpolation. *Neural Computation*, 4:415–447, 1991. 104

[93] M. Markou and S. Singh. Novelty detection: A review - part 1: Statistical approaches. *Signal Processing*, 83:2003, 2003. 41

[94] M. Markou and S. Singh. Novelty detection: A review - part 2: Neural network based approaches. *Signal Processing*, 83, 2003. 26, 34, 39

[95] S. Marsland. Novelty detection in learning systems. *Neural Computing Surveys*, 3:157–195, 2003. 39

[96] D. Martinez. Neural tree density estimation for novelty detection. *IEEE Transactions on Neural Networks*, 9(2):330–338, 1998. 39

[97] A. S. Mian, M. Bennamoun, and R. Owens. Three-dimensional model-based object recognition and segmentation in cluttered scenes. *IEEE Transactions in Pattern Analysis and Machine Intelligence*, 28(10):1584–1601, 2006. 44

[98] R. Miikkulainen and M. G. Dyer. Natural language processing with modular neural networks and distributed lexicon. *Cognitive Science*, 15:343–399, 1991. 98

[99] K. Mikolajczyk and C. Schmid. Scale & affine invariant interest point detectors. *International Journal of Computer Vision*, 60(1):63–86, 2004. 63, 130, 134

[100] K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 27(10):1615–1630, 2005. 46

[101] A. Mohr and M. Gleicher. Non-invasive, interactive, stylized rendering. In *Symposium on Interactive 3D Graphics*, pages 175–178, 2001. 145

[102] F. Mokhtarian and A. K. Mackworth. A theory of multiscale, curvature-based shape representation for planar curves. *IEEE Transcations on Pattern Analysis and Machine Intelligence*, 14(8):789–805, 1992. 92

[103] D. S. Moore. Tests of chi-square type. In R. B. D'agostino and M. S. Stephen, editors, *Goodness-of-Fit Techniques*. Marcel Dekker, New York and Basel, 1986. 62

[104] Choudhury R. K. Mulchrone, K.F. Fitting an ellipse to an arbitrary shape: implications for strain analysis. *Journal of Structural Geology*, 26(1):143–153, 2004. 92

[105] R. Ohbuchi, K. Osada, T. Furuya, and T. Banno. Salient local visual features for shape-based 3d model retrieval. In *Shape Modeling International*, pages 93–102, 2008. 44

[106] M. E. Otey, A. Ghoting, and S. Parthasarathy. Fast distributed outlier detection in mixed-attribute data sets. *Data Min. Knowl. Discov*, 12:2–3, 2006. 40

[107] J. Owens and A. Hunter. Application of the self-organising map to trajectory classification. In *EE Visual Surveillance Workshop*, Dublin, Ireland, 2000. 103

[108] Z. Pan, X. Wei, and J. Yang. Geometric model reconstruction from streams of directx 3d game application. In *2005 ACM SIGCHI International Conference on Advances in computer entertainment technology (ACE '05)*, pages 242–245, New York, NY, USA, 2005. ACM. 145

[109] K. Pearson. Notes on the history of correlation. *Biometrika*, 13(1):25–45, 1920. 100

[110] J. Petit and R. Brémond. A high dynamic range rendering pipeline forinteractiveapplications. *The Visual Computer*, 26:533–542, 2010. 10.1007/s00371-010-0430-5. 20

[111] M. Pettersson. A compiler for natural semantics. In *6th International Conference in Compiler Construction*, pages 177–191, 1996. 152

[112] Matt Pharr, editor. *GPU Gems 2 - GPU Flow-Control Idioms*, chapter 34, pages 547–555. Addison-Wesley Professional, 2005. 49

[113] D. Pinelle, N. Wong, and T Stach. Heuristic evaluation for games: usability principles for video game design. In *Twentysixth annual SIGCHI conference on Human factors in computing systems*, pages 1453–1462. ACM, 2008. 14

[114] A. R. Pope. Model-based object recognition a survey of recent research. In *USENIX Technical Conference*, 1994. 43

[115] G. Potamianos, C. Neti, and J. Luettin. Audio-visual automatic speech recognition : An overview. *Robotics*, pages 1–30, 2004. 32

[116] S. Rajesh, S. Prathima, and L. S. S. Reddy. Unusual pattern detection in dna database using kmp algorithm. *International Journal of Computer Applications*, 1(22):1–5, February 2010. Published By Foundation of Computer Science. 32

[117] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. *SIGMOD Rec.*, 29:427–438, May 2000. 40

[118] S. Roberts and L. Tarassenko. A probabilistic resource allocating network for novelty detection. *Neural Comput.*, 6:270–284, March 1994. 41

[119] R. Rojas. *Neural Networks: A Systematic Introduction*. Springer, 1 edition, July 1996. 37

[120] G. Rosado. *GPU Gems 3 - Motion Blur as a Post-Processing Effect*, chapter 27, pages 575–581. Addison-Wesley Professional, 2005. 49

[121] P. M. Roth and M. Winter. Survey of appearance-based methods for object recognition. Technical report, Inst. for Computer Graphics and Vision, Graz University of Technology, Austria, 2008. 45

[122] B. Ruppert. New directions in virtual environments and gaming to address obesity and diabetes: Industry perspective. *Journal of Diabetes Science and Technology*, 5(2):277–282, March 2011. 2

[123] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003. 36, 41, 98

[124] D. Scherzer, M. Wimmer, and W. Purgathofer. A survey of real-time hard shadow mapping methods. *Computer Graphics Forum*, 30(1):169–186, Feb 2011. 16, 128

[125] D.l Scherzer, M. Wimmer, and W. Purgathofer. A survey of real-time hard shadow mapping methods. In Helwig Hauser and Erik Reinhard, editors, *EG 2010 - State of the Art Reports*, pages 21–36, Norrköping, Sweden, 2010. Eurographics Association. 21, 127

[126] Noor Shaker, Georgios Yannakakis, and Julian Togelius. Towards automatic personalized content generation for platform games. *Computer*, (Hudlicka 2008), 2009. 6

[127] C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine (Series 7)*, 41(314):256–275, 1950. 4

[128] L.G. Shapiro and G.C. Stockman. *Computer Vision*. Prentice Hall, Upper Saddle River, New Jersey, 2001. 61

[129] W. A. Shewart. *Economic control of Quality of Manufactured Product*. Van Nostrand Reinhold Co., New York, 1931. 41

[130] M. Stommel. Binarising sift-descriptors to reduce the curse of dimensionality in histogram-based object recognition. *International Journal of Signal Processing, Image Processing and Pattern Recognition*, 3(1):25–36, March 2010. 45, 91

[131] M. J. Swain and D. H. Ballard. Color indexing. *International Journal of Computer Vision*, 7:11–32, 1991. 62

[132] László Szirmay-Kalos and Tamás Umenhoffer. Displacement mapping on the gpu - state of the art. *Comput. Graph. Forum*, 27(6):1567–1592, 2008. 16, 20

[133] P. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. 39

[134] R. T. Tan, K. Nishino, and K. Ikeuchi. Separating reflection components based on chromaticity and noise analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1373–1379, October 2004. 118

[135] S. Theodoridis and K. Koutroumbas. *Pattern Recognition, Fourth Edition*. Academic Press, 4th edition, 2008. 31, 32

[136] C. Thurau, T. Paczian, G. Sagerer, and C. Bauckhage. Bayesian imitation learning in game characters. *Int. J. Intelligent Systems Technologies and Applications.*, 2:284–295, February 2007. 144

[137] J. Togelius, R. De Nardi, and S. M. Lucas. Towards automatic personalised content creation in racing games. In *IEEE Symposium on Computational Intelligence and Games*, 2007. 6

[138] A. Toshev, A. Makadia, and K. Daniilidis. Shape-based object recognition in videos using 3d synthetic object models. In *CVPR*, pages 288–295, 2009. 44

[139] C. van der Walt and E. Barnard. Data characteristics that determine classifier performance. pages 166–171, Nov 2006. 33

[140] V. N. Vapnik. *Statistical Learning Theory*. Wiley-Interscience, September 1998. 39

[141] J. H. Victoria and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, October 2004. 41

[142] P. Vorderer, C. Klimmt, and U. Ritterfeld. Enjoyment: At the heart of media entertainment. *Communication Theory*, 14(4):388–408, November 2004. 3

[143] K.Q. Weinberger, J. Blitzer, and L. K. Saul. Distance metric learning for large margin nearest neighbor classification. In *Neural Information Processing Systems*. MIT Press, 2006. 39

[144] L. Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Computer Graphics*, 12:270–274, August 1978. 21, 27

[145] L. Williams. Pyramidal parametrics. *SIGGRAPH Computer Graphics*, 17:1–11, July 1983. 21

[146] E. Wolfgang. *Shader X4: Advanced Rendering Techniques*. Charles River Media, Inc, Hingham, MA, 2006. 128, 129

[147] A. Woznica and A. Kalousis. A new framework for dissimilarity and similarity learning. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 386–397, 2010. 40

[148] J. Wright, Y. Ma, J. Mairal, G. Sapiro, T. S. Huang, and S. Yan. Sparse representation for computer vision and pattern recognition. *Proceedings of The IEEE*, 98:1031–1044, 2010. 32

[149] G. Xiao, F. Southey, R. C. Holte, and D. Wilkinson. Software testing by active learning for commercial games. In *Proceedings of the 20th national conference on Artificial intelligence - Volume 2*, pages 898–903. AAAI Press, 2005. 6, 13

[150] Y. Yang, J. Zhang, J. Carbonell, and C. Jin. Topic-conditioned novelty detection. In *Knowledge Discovery and Data Mining*, pages 688–693, 2002. 40

[151] G. N. Yannakakis and J. Hallam. Towards optimizing entertainment in computer games. *Applied Artificial Intelligence*, 21:933–971, November 2007. 7

[152] Y. H. Yee, S. N. Pattanaik, and D. P. Greenberg. Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments. *ACM Trans. Graph.*, 20(1):39–65, 2001. 159

[153] A. Ypma, E. Ypma, and R. P.W. Duin. Novelty detection using self-organizing maps. In *ICONIP'97*, pages 1322–1325. Springer, 1997. 101

[154] M. Yuan, F. Farbiz, C. M. Manders, and K. Y. Tang. Robust hand tracking using a simple color classification technique. In *VRCAI '08: Proceedings of The 7th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, pages 1–5, New York, NY, USA, 2008. ACM. 92

[155] B. D. Zarit, B. J. Super, and F. K. H. Quek. Comparison of five color models in skin pixel classification. In *RATFG-RTS '99: Proceedings of the International Workshop on Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems*, page 58, Washington, DC, USA, 1999. IEEE Computer Society. 93

[156] W. Zhang, L. He, Y. Deng, J. Liu, and M. T. Johnson. Time-frequency cepstral features and heteroscedastic linear discriminant analysis for language recognition. *IEEE Transactions on Audio, Speech & Language Processing*, 19:266–276, 2011. 32