VISUAL ATTENTION IN DYNAMIC ENVIRONMENTS AND ITS APPLICATION TO PLAYING ONLINE GAMES

IULIIA KOTSERUBA

A THESIS SUBMITTED TO THE FACULTY

OF GRADUATE STUDIES IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

YORK UNIVERSITY

TORONTO, ONTARIO

April 2016

Abstract

In this thesis we present a prototype of Cognitive Programs (CPs) - an executive controller built on top of Selective Tuning (ST) model of attention. CPs enable top-down control of visual system and interaction between the low-level vision and higher-level task demands.

We implement a subset of CPs for playing online video games in real time using only visual input. Two commercial closed-source games - Canabalt and Robot Unicorn Attack - are used for evaluation. Their simple gameplay and minimal controls put the emphasis on reaction speed and attention over planning.

Our implementation of Cognitive Programs plays both games at human expert level, which experimentally proves the validity of the concept. Additionally we resolved multiple theoretical and engineering issues, e.g. extending the CPs to dynamic environments, finding suitable data structures for describing the task and information flow within the network and determining the correct timing for each process.

Acknowledgments

First and foremost I would like to express my sincerest gratitude to my supervisor Prof. John Tsotsos for giving me freedom to work on this topic and for his support, patience and helpful suggestions throughout my thesis writing process. I could not wish for a better supervisor.

I also would like to thank committee members, Prof. Minas Spetsakis, Prof. Thilo Womelsdorf and Prof. Melanie Baljko for their time and effort in reviewing my work.

I am especially grateful to my colleagues Amir Rasouli, Eugene Simine and Calden Wloka for useful discussions.

Finally, but not least, I would like to thank my parents for their love, encouragement and support.

Contents

List of Tables

List of Figures

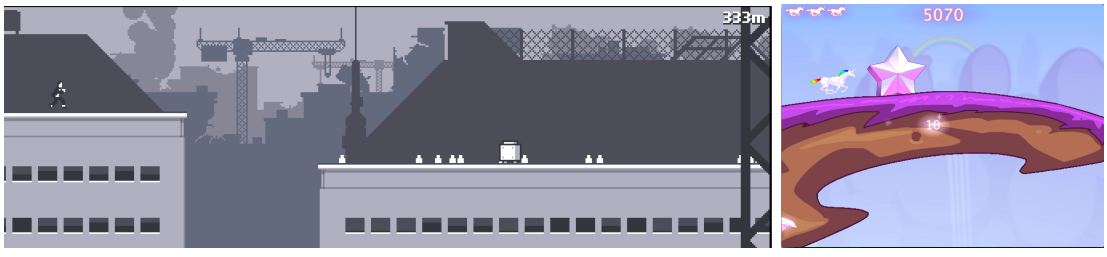
List of Abbreviations

AIM	bottom-up saliency algorithm Attention based on Information Maximization
BB	Blackboard
cFOA	center of Focus of Attention
CNN	Convolutional Neural Network
CPs	Cognitive Programs
FC	Fixation Control
FOA	Focus of Attention
HBPM	History Biased Priority Map
mLTM	Long Term Memory for methods
ppi	pixels per inch or pixel density
PPM	Peripheral Priority Map
RANSAC	RANdom SAmple Consensus
ST	Selective Tuning model of visual attention
tWM	Task Working Memory
vAE	Visual Attention Executive
VH	Visual Hierarchy
VR	Visual Routines
vTE	Visual Task Executive
vWM	Visual Working Memory

1 Introduction

1.1 Motivation

The main goal for this thesis was to develop a working prototype for Cognitive Programs (CPs) [85] - an executive controller built on top of the model of visual attention enabling it to perform non-trivial dynamic visually guided tasks. CPs are inspired by the concept of Ullman's Visual Routines [86] combined with recent findings about the neurophysiology of visual attention. The visual attention module is based on Selective Tuning (ST) [84] - a biologically plausible model of visual attention, which has been successfully implemented and whose predictions were strongly supported by experimental data. Its complex hierarchical system mimics human vision and allows for both top-down and bottom-up processes to influence visual processing. The purpose of Cognitive Programs is to control the execution of ST by modifying the way it treats inputs based on the current visual task and directing outputs to appropriate parts of the framework. Selective Tuning and Cognitive Programs are designed to handle a broad range of visual inputs. In order to demonstrate the capabilities of Cognitive Programs we implemented parts of it needed for playing video games. The reasons for choosing this particular task are twofold: 1) playing a game requires a significant amount of visual analysis, for instance visual search, tracking, object localization, recognition and classification, etc., 2) modern video games provide a controlled and visually rich environment for testing the new system.



(a) Canabalt (b) Robot Unicorn Attack

Figure 1: Screenshots of the games

Two games selected for the purposes of this demonstration were Canabalt and Robot Unicorn Attack (Figure 1). Both are among the most played representatives of a new subgenre of platform games called endless scrollers, and have many characteristics which make them suitable for testing a dynamic model of attention. The maps for games of this type are procedurally generated making every new session different from the previous one. The objective is to run for as long as possible while avoiding various obstacles. Both games have simple controls: in Canabalt the character jumps over an obstacle when the 'X' key is pressed, while in Robot Unicorn Attack pressing 'X' or 'Z' makes the character jump or dash through the obstacle respectively. The map scrolls forward automatically. The score depends on the distance traveled and sometimes can be increased by collecting special objects. Typically, the speed of the games gradually increases, making quick reaction, attention and precision more important than planning.

Both of these games are commercial and no source code or API is available, hence, two additional problems had to be addressed. First, no direct access to noise-free information from the game engine was possible. This meant that all decisions must be made based only on what is visible on the screen. Second, since the games cannot be artificially slowed down, all processing must be done in real time. These characteristics, though challenging, make these games an ideal test for our models.

Using games as a visual input is not a new idea and first such attempts were made in the 1980s. They were not very successful due to the inadequate computing resources available at that time for the amount of processing required to accomplish such tasks. Another issue was in the selection of the games for research. Many authors favored the exploration type games, that required a player to interact with various objects on the screen and solve puzzles. The demand for interaction with multiple objects shifted the focus from vision to developing complex gameplay strategies and resolving semantic problems. We tried to avoid some of these issues by selecting games with fewer objects and simple gameplay. By doing so we focused more on visual attention and thus the feasibility of the CPs concept. Finally, designing an algorithm for playing games in real time not only demonstrates the importance of optimization

for visual tasks, but also utilizes recent advances in general purpose GPU programming.

1.2 Outline

The **second chapter** of this thesis provides an overview of research background. Building an algorithm to visually play video games requires developments from disparate areas of research: computer vision, visual attention, cognitive architectures, game AI and GPU computing. We first give a short overview of how sensory input is handled in the established cognitive architectures, followed by a discussion of various visual attention models based on visual routines. We then introduce Cognitive Programs as a controller for a modern model of visual attention and finally review some relevant works in the area of game AI for platform games.

The **third chapter** contains the technical details of the elements of Cognitive Programs for the task of playing online games and discusses modifications that are necessary to enable real-time performance. Here we also examine practical issues stemming from working with real-time data. Finally we review tools used for learning game physics, visual debugging and gathering statistics about the games.

Chapter four evaluates the performance of the visual system and overall performance of the framework. To our knowledge no other software agent can play our games or similar games in real time based on vision alone, therefore we only compare our results with those of human expert players. Since both games used in this research have a large number of followers, numerous recorded scores are available online. We also discuss different aspects of the algorithm affecting the final game score, e.g. the accuracy of speed estimation and object detection.

Finally, in **chapter five** we give a summary of the thesis along with what was learned and suggestions for future research.

2 Related research

Playing video games in real time using only visual input requires developments from several areas of research: computer vision, visual attention, cognitive architecture design, game AI and general purpose GPU programming. Unfortunately, these fields, although conceptually tightly related, in practice do not intersect much. Most cognitive architectures focus on high cognitive functions and either avoid the perceptual side of human cognition or place their agents in simulated worlds. On the other hand, visual attention research predominantly uses real-world data to build models, which are often limited in scope to explaining separate stages of visual processing and leave higher order processes out.

Another aspect of visual processing is attention. Even when not explicitly stated, some elements of attention are commonly used to reduce the computational load, e.g. selection of areas of interest or feature selection. Attentional top-down guidance has only been implemented for a few relatively simple visual operations like visual search. Cognitive Programs is a framework, which provides a mechanism to extend top-down control and tuning of visual system to more general visual tasks. An approach like this could be useful to achieve attentive perception within cognitive architectures.

2.1 Vision in Cognitive Architectures

Traditionally, cognitive architectures study higher cognitive functions, such as planning, reasoning, grammar comprehension, etc. These processes are sequential in nature and rely on symbolic manipulation and complex knowledge structures. Motor control, perception, memory retrieval and other lower cognitive functions, although acknowledged by many researchers as being equally important, are often not a focus of research. For instance, out of 31 established cognitive architectures listed on cogarch.org and bicasociety.org two thirds focus on higher-level cognition and either do not accept any perceptual input at all or work only in simulated environments (Table 1).

Architectures that do not implement perception usually focus on general cognitive abilities,

knowledge representation, memory organization, learning and complex planning of several concurrent tasks. Typical applications of such systems are playing turn-based games (ticktacktoe, chess, etc.), solving puzzles, developing battlefield strategies, categorical reasoning, etc. On the other hand, nearly all cognitive architectures that use the real sensory data, are designed to perform tasks, which do not require a lot of symbolic processing, such as navigation, object recognition, object tracking, and visual search.

Years active	Name	Perception	References
1979-present	4D-RCS	real	[2, 3]
1982-2007*	CAPS	no perception	[34]
1983-2003	AIS	simulation and real	[29]
1983-present	Soar	simulation	[91]
1985-present	Subsumption	real	[9]
1988-1991*	RALPH	simulation	[54, 73]
1988-1997	Prodigy	no perception	[89]
1988-2010*	Disciple	no perception	[79, 80]
1990-1991*	Homer	simulation	[90]
1990-1991*	Teton	no perception	[88]
1990-1992*	ERE	no perception	[15]
1991-1992*	ATLANTIS	simulation	[?]
1992-present	Chrest	no perception	[40]
1993-present	GLAIR	simulation and real	[76]
1993-present	ACT-R	simulation	[4]
1994-present	FORR	no perception	[?]
1996-2013*	CLARION	simulation	[78]
1996-2014*	LEABRA	real	[56]
1997-2011*	ICARUS	simulation	[42, 41]
1997-present	EPIC	simulation	[37]
2001-2008*	REM	no perception	[52, 51]
2002-2006*	Polyscheme	real	[12, 11]
2004-2012*	Companion	no perception	[20]
2005-2010*	GMU BICA	no perception	[74]
2005-present	NuPIC	no perception	[28]
2006-present	LIDA	simulation	[22, 21]
2008-present	OpenCogPrime	simulation and real	[25]
2008-present	BECCA	real	[69, 68]
2009-present	CERA-CRANIUM	simulation and real	[5]
2009-present	CoJACK	simulation	[16]
2010-2013*	ADAPT	real	[6, 45]

Table 1: List of cognitive architectures in chronological order. * marks the year of the latest activity (paper publication or code update). Architectures implementing perception of real world data are highlighted.

Existing systems can be split into three groups based on how they treat perceptual input: 1) architectures built around perception; 2) architectures using sensory data directly; and 3) architectures treating perception as a "black box". Below we discuss these groups in more detail.

First group includes architectures capable of autonomous operation in real environments. As a result, these systems can efficiently process real-world data at real-time, but are very limited in their cognitive ability and application. Since the perceptual component is dominant, the rest of the components are built around it. For example, consider two cognitive architectures of this type: RCS ([3, 2]) and ADAPT ([6, 45]).

RCS is designed for autonomous driving and implements a layered architecture. The lowest level performs reflex actions (e.g. stop if you hit an object) and is directly connected to the physical sensors. Higher levels of the hierarchy work with more abstract data derived from the sensor readings, such as various maps and features. Finally, at the top level the global planning occurs. An approach like this avoids the problem of translating sensory information to the symbolic domain, but makes it harder to do general reasoning.

ADAPT architecture enables autonomous navigation for the mobile robot. This system follows a different path of integrating sensing into cognition. It creates a simulated copy of the world and at every cognitive cycle focuses on the discrepancies between the new sensory data and the internal model. The properties of the objects can be obtained from the internal model and passed to the traditional cognitive architecture - Soar [91] in this case. ADAPT uses OpenCV and Kinect for visual processing. It also builds a 3D model of the environment using an open-source graphics engine Ogre3D. Finally, the NVIDIA PhysX SDK is applied to model the physics of the scene. Since maintaining a detailed consistent internal model is computationally expensive, active vision and visual attention mechanisms (e.g. limiting processing to regions with the higher uncertainty) are employed to limit the amount of processing. It is especially important for quick action in dynamic scenarios (e.g. tracking and intercepting fast moving objects).

Cognitive architectures in the second group use sensory information directly. For example,

in Subsumption architecture [9] raw sensor output was connected directly to actions without any internal world representation. Even though it led to a range of interesting complex behaviors, it is a general consensus that architectures based on this idea are not suitable for most activities involving planning or reasoning.

More recent architectures of this kind use neural networks to mimic the structure of the human brain. For example, LEABRA [56] is based on the anatomy of the ventral pathway of the brain (V1, V2/V4, IT) and implements it as a hierarchy of filters. The responses at the highest level are connected to semantic concepts and the whole system can be used for object recognition. Another example, BECCA ([68, 69]), also uses a hierarchy of filters to build features and then applies reinforcement learning to find actions (associated with features) that give the highest reward for a particular task (e.g. recognition, focusing on particular objects in the scene).

Although these approaches are conceptually simple and biologically plausible, it is harder to extend them to perform complex actions compared to symbolic systems. So far systems using direct perception are limited to tasks like recognition or visual tracking and work only with low-resolution synthetic data.

Finally, a third group of architectures treats perception as an isolated module responsible for translation of incoming sensory data into symbols required for higher cognitive functions. This approach is typical for architectures operating in simulated environments. As we have already seen from architectures in the first group, in order to work with complex real-world data, perception must be tightly integrated with the rest of the system. This means that moving an architecture from a simulated domain to a real world environment is likely more involved than replacing one 'black box' with a more sophisticated one.

A common workaround to this problem is to use a set of independent modules to process the same sensory data using different methods in parallel and then combine the results to increase robustness to noise. For example, the Polyscheme [12, 11] architecture splits cognition into a predefined number of separate modules called specialists, where each analyzes incoming data independently using arbitrary algorithms and data structures. At each cognitive cycle, all

specialists transform their data to a single common language, which is then used for more general reasoning. As a result, embodied Polyscheme agents can follow a bright object in the scene by applying standard vision techniques (color segmentation, stereo vision and optical flow).

The main problem of all systems described above is noise both from the sensors and from the changes in the environment. One way to address it is to work in non-cluttered, high-contrast environments with simple objects. For example, ADAPT relies on edge detection and thresholding in order to detect a bright colored ball bouncing off a wall and makes an assumption that similar features detected in a different frame belong to the same object. However, a small change in the illumination conditions is likely to break this system because it does not have any adaptation mechanisms to compensate for dynamic conditions.

Architectures like RCS and Polyscheme attempt to resolve the problem of noisy data by accumulating information from many different types of sensors (e.g. vision, sonars, lasers, etc). Biologically inspired architectures such as BECCA and LEABRA by design have the ability to adapt and learn, however, their performance has been only shown on synthetic data.

Besides noise, the volume of data incoming through sensors can also pose a serious problem, especially for time critical applications like autonomous driving or object tracking. The common approach to reducing the amount of data is to limit it to a certain area (region of interest) or range (thresholding).

2.2 Ullman's Visual Routines

The theory of Visual Routines (VR) [86] introduced by Ullman in 1984 conceptualizes visual perception of spatial properties and relations between objects as a complex hierarchy of processes rather than immediate result of a single operation. He suggests that vision could be reduced to a series of atomic context-independent operations assembled for a particular task. Influenced by Marr's theory of vision [46] he describes vision as a two-stage process. First, the base representation is created from the incoming data in a bottom-up fashion. Second, various operations are applied sequentially to the base representation to solve a particular task. These

operations are called visual routines.

In order to implement visual routines as outlined in the paper the following components are required:

- *Base representation* is a result of the bottom-up processing of the data (primal sketch and 2½-D sketch in Marr's terms). This representation has the following characteristics: it is unarticulated, viewer-centered, uniform and bottom-up driven. It also contains local descriptions of depth, orientation, color and direction of motion at a point.
- *Incremental representations* are the results of applying visual routines to the base representation.
- *Atomic operations* are the most general operations, which can be applied to any location within the base or intermediate representations. Possible candidates include shift of processing focus, indexing (defining next targets for focusing), marking (memorizing locations for future use), boundary tracing and determining inside/outside relations.
- *Assembly, execution, and storage of visual routines.* Visual routines for common tasks (e.g. object recognition) should be stored in a skeleton form and parametrized during runtime. Some goals may require assembly of the routine from scratch. Execution of the routines relies on visual attention, however, particular implementation is not discussed.

Since the exact mechanics of these elements in human visual system are not known, the biological plausibility of the algorithms implementing them is not important as long as the final result is the same.

An ability to control the focus of attention is essential for a functional system of visual routines. Shift of focus allows the application of the same routines to different locations and limits the processing to a small region of space. This shift operation depends on indexing, which marks conspicuous locations by an indexable property, usually a combination of features computed in the early stages, such as color, motion, orientation, disparity, etc. A hierarchical network with connections between different layers and elements within individual layers is suggested as a possible implementation of execution and control of attention. Additionally, tasks like counting and visual search cannot be done without a map, where salient and al-

ready visited locations would be saved and masked to prevent the system from counting some elements twice.

Ullman cites many psychophysiological and neurophysiological studies to support validity of visual routines. In the past three decades our understanding of how brain works has seen significant changes, but the core concept of visual routines has not been disproved. For example, [67] provides evidence from neurophysiological studies that serial cognitive tasks may be implemented by networks of neurons spanning several areas of the cerebral cortex. Yi and Ballard [94] test the validity of visual routines by building a simulator to perform visuomotor tasks of pouring coffee or making a peanut butter sandwich and comparing it to 3D eye-tracking data obtained from human subjects. Even though data showed high variability, it was still possible to express their actions as a chain of subtasks, model it as a Markov process and find correspondence with the simulated results.

Hayhoe [30] examined the phenomenon of "change blindness" and conducted experiments to prove that the visual system represents only the information, which is necessary for the immediate visual tasks. In one experiment the subjects were asked to reproduce a pattern of colored blocks. They made eye movements to the model pattern, sometimes looking at the same block twice within a short interval, clearly preferring it to using visual memory. An explanation was offered that the first saccade was to determine the color and the second saccade was to find the relative location of the block in the pattern.

No prototype of visual routines was provided by Ullman and the original publication was more of a program paper justifying a need for such system. As a result, every implementation of the visual routines follows its own interpretation of the theory and fills in missing elements. Table 2 provides short descriptions of the projects based on visual routines. Since visual routines are an intermediate step between the early visual representations and higher level components of the visual system, they can be used even in simulated environments. For example, an algorithm called Pengi [1] applies visual routines to play the popular SEGA game Pengo (Figure 2a), which involves navigation of a penguin in a 2D maze populated with killer bees and ice cubes. Since coordinates and properties of all objects in the screen are readily

Pengi (1987) [1]
Task: playing SEGA video game Pengo
Base representation: game simulator state
Atomic operations: N/A
Visual routines: updating locations of enemies; checking whether a kicked wall block will collide with anything
Execution: predefined

VR Framework (1988) [71, 70]
Task: reason about properties of simple 2D shapes
Base representation: color channel map, edge map, disparity map
Atomic operations: register I/O operations, spreading activation
Visual routines: inside, outside, connected, part of, is-vertical, is-closed, is-concave, is-triangle, is-dot, etc.
Execution: Triggered by evaluating a VRL query

Sonja (1990) [13]
Task: playing computer game Amazon
Base representation: game simulator state
Atomic operations: N/A
Visual routines: measuring distances, directions and angles; tracking; coloring; edge following
Execution: emergent

ALIVE (1993) [32, 33]
Task: find left and right hand on a silhouette of a human
Base representation: segmentation map, top edges, bottom edges, right edges, left edges
Atomic operations: N/A
Visual routines: add and subtract points, find-bottom-edge, find-top-edge, leftmost-point, average-point
Execution: genetic programming

SKETCHY (1995) [61]
Task: reasoning about graphs
Base representation: simulator state
Atomic operations: N/A
Visual routines: examine label, coordinate-at-point, is-right-of, is-left-of, is-above, is-below, intersects, touches
Execution: predefined

Active vision (1995)[64]
Task: identify and locate 3D objects in a real environment
Base representations: multi-scale steerable Gaussian filter responses
Atomic operations: I/O operations, disparity calculation
Visual routines: computing and comparing "zip-codes" for objects, object identification and localization
Execution: predefined

Jeeves (1995) [31]
Task: spatial reasoning about colored cubes
Base representation: input 64×29, color, intensity, temporal and spatial derivatives, Laplacian, edges
Atomic operations: edge detection, figure-ground and color segmentation
Visual routines: is-green, is-blue, is-red, is-above, is-below, vertical, horizontal
Execution: triggered by evaluating user queries

Driving Simulator (1996) [48, 49]
Task: driving a car in a simulated environment
Base representation: simulator state
Atomic operations: N/A
Visual routines: hear-horn, gaze-object, gaze-direction, gaze-speed, gaze-distance, gaze-color
Execution: reinforcement learning

AV-Shell (1996) [17]
Task: visually controlling robotic arm
Base representation: edges, segmentation, laplacian, depth, optical flow
Atomic operations: edge detection, figure ground segmentation
Visual routines: active contours (snakes), template recognition, fixation, pursuit, saccade, focus adjustment, motion detection
Execution: triggered by evaluating an Robot Schema expression

Tactical driving (1998) [24]
Task: driving a car in a simulated environment
Base representation: color channels, multi-scale steerable DoG filter responses, optical flow
Atomic operations: N/A
Visual routines: traffic light detection, stop sign detection, car following, obstacle avoidance
Execution: predefined

VR and Attention (1995)[65]
Task: tracking objects, following directions on where to look
Base representation: Gaussian filters, optical flow, color saliency
Atomic operations: figure_ground_motion, get_size, get_orientation, match_regions, select_color, select_motion
Visual routines: find human, find human arm, select region in the scene, track object, wait for object
Execution: emergent

Tekkotsu and AIBO (2000) [26, 83]
Task: playing tic-tac-toe
Base representation: image from the camera
Atomic operations: morphological operations, connected components, region filling, boundary calculation
Visual routines: find board lines, find board boundaries, find cells, determine cell occupancy
Execution: predefined

Table 2: A chronological list of past projects implementing visual routines

available from the simulator, the atomic operations simply retrieve this data and use it for spatial reasoning (measuring angles, distances, directions and ray-following).

One of the earliest and most complete implementations of visual routines is VR Framework [71, 70], which was designed to reason about properties of simple 2D shapes in 32×32 synthetic grayscale images. The base representation is a set of retinotopic maps with color, edge and disparity information. A total of 36 visual routines are formulated to cover all possible relations in a limited environment of 2D shapes, e.g. inside/outside, is-connected, is-part-of, is-vertical, etc. These routines are composed of atomic operations, most common of which are various I/O operations on registers, spreading activations over maps, clearing and composing maps. Appropriate routines are triggered by execution of queries in a specially defined Visual Routine Language, such as 'How many vertical lines are in the image?'.

Rao and Ballard [64] developed an active vision system based on the idea of visual routines to identify and locate objects. Their setup consists of a binocular head with two color cameras taking images of size 512×480 at 30 fps. Their base representation contains results of convolving the camera image with 9 different 8×8 discrete Gaussian derivative kernels on 5 scales. It also includes a figure-ground segmentation map obtained by zero disparity filtering. Normalized multi-scale filter response vectors (also called "zip-codes") are used for object localization and identification. First, a database of filter responses for 72 different objects is created (36 images for each object at 10° rotational increments in pose). In order to locate a particular object its "zip-code" from the database is compared to "zip-codes" of each point within the area in the camera image masked by the figure-ground segmentation. Locations matching the predefined object are marked with a cross. Similarly, to identify an unknown object in the camera, its "zip-code" is compared to the models in the database. The system performs in real time due to hardware-accelerated convolution and is capable of recognizing 70% of the test cases using only one point at the object centroid and up to 100% when the number of points is increased to 25.

Another system based on visual routines is developed by Rao [65] to track moving objects and follow directions on where to look for an object. The particular setup, camera, resolution of

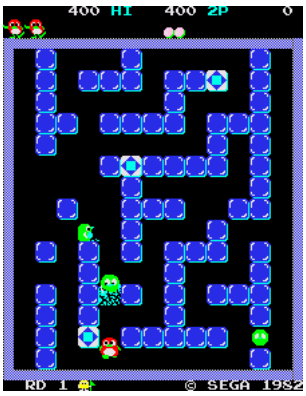
the images, etc. are not stated. The system works with real images and its base representation consists of responses to steerable filters, color maps, blob detector and optical flow. A large set of elementary operations is defined to establish properties and spatial relations between the objects in the scene, e.g. *figure_ground_motion*, *get_size*, *get_direction*, *select_color*, *select_motion*, *select_blob*, etc.

Typically, when an actual visual input is involved, extra processing is required to compensate for the noise and uncertainty introduced by the sensors. Hence, projects dealing with real images operate in structured environments with uniquely color-coded objects (e.g. AIBO playing ticktacktoe [26, 83] shown in Figure 2d or Jeeves [31] reasoning about bright colored blocks placed on a uniform background). Typically, the base representation includes multiple color channels, spatial and temporal derivatives, edge maps, optical flow and segmentation maps. Elemental operations, such as morphological functions, motion detection, and edge following are applied to the base representation to solve a task.

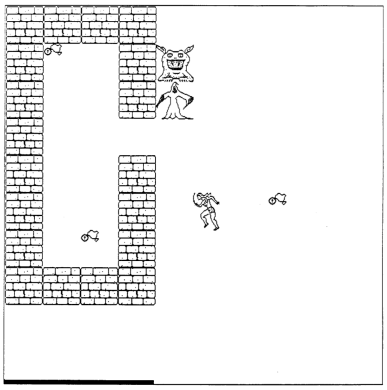
The implementations of visual routines so far assumed a fixed set of elementary operations and predefined routines. In fact, having a database of predefined visual routines is the most common approach, albeit labor intensive and not very flexible (in Pengi about 30 visual routines were implemented in 1000 lines of code and covered only a subset of possible gameplay situations).

Chapman and Agre propose to avoid the assembly and storage problem by reinterpreting the term "routine" as an emergent pattern. Their system Sonja [13] plays a computer game called Amazon (Figure 2b). It has a two-way connection to all visual operators and is responsible for providing them with the arguments, essentially merging late vision with cognition.

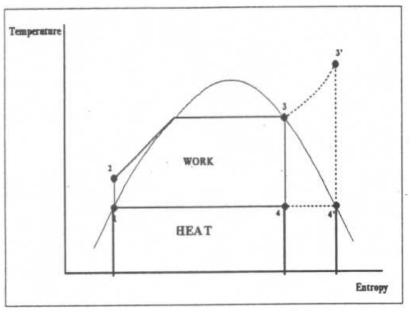
Rao [65] follows up on the emergent visual routines and provides data showing the validity of such approach. He collects traces of the state of his program when observing a falling/bouncing ball and clusters them. He then proposes to average the local spatial contexts of the similar traces to get a template, in this case for "the ball falling to the right". However, it is not clear whether the learned routine actually works, since no demonstration of its performance is provided. The working examples in Rao's paper are based on predefined visual routines (finding



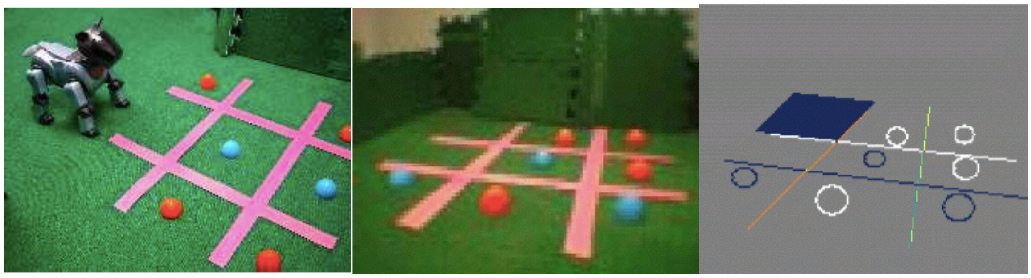
(a) Pengo



(b) Amazon



(c) SKETCHY



(d) AIBO playing Tic-Tac-Toe (left to right: environment, AIBO's view, processing result)

Figure 2: Examples of projects using visual routines

human in the scene and tracking a moving ball as it passes behind an obstacle). Interpretation of visual routines as emergent patterns is rather uncommon and majority of implementations treat routines as 'program fragments'.

A different way of solving the assembly problem is to reformulate a vision problem as a Prolog-like query. When this query is evaluated, each part of the query calls an appropriate visual routine. For example, in Jeeves [31] this approach is used to compute 2D spatial relations between colored cubes. The queries are implemented as Horn clauses and the visual system acts as a simplified Prolog engine. For example, a query 'find the red cube' is equivalent to finding an image region satisfying a conjunction of primitive features 'red' and 'cube', which have predefined visual routines. Initially, all regions satisfying the first feature are serially enumerated and then tested for the other feature. Failure of any test causes backtracking

implemented as a return inhibition map.

In AV-Shell [18] visual routines are used as the building blocks for vision-based tasks in situated robotics. Composition of elemental operations into complex activities is achieved by traversing a parse tree built from an expression in Robot Schemas [44] notation. For example, pursuit is a continuous perception-action process composed of vergence control, foveal motion detection and dynamic accommodation; they execute in parallel and are combined, provided that all complete successfully.

Most researchers acknowledge that predefining visual routines for every task is time-consuming and inefficient, but very few develop automatic techniques for composing routines. For example, McCallum applied reinforcement learning (RL) to learn visual routines for the task of navigating a car between trucks while avoiding collisions [49]. The main obstacle for learning algorithms is the high dimensionality of the search space, for example with only 5 possible actions and 7 sensor readings the environment has over 21,000 world states and over 2,500 sensor states. With a short-term memory of size 3 an internal agent space grows to 2500^3 states. Author proposes a new optimization technique called U-Tree [48] to reduce the search space to 143 states. A U-Tree is built during the training phase: it starts by recording raw data (action-percept-reward triple) and selectively adding branches when additional distinctions are needed. The leaves of the tree store Q-values of the RL agent. At runtime when an observation is received, the current internal state can be determined by following branches starting with the root until a leaf node is reached. Such a structure implements both feature selection and short-term memory (hidden state). Experiments show that the learned policy outperforms the random method by 77%. Despite the fact that dimensionality reduction methods like U-Tree improve the speed and robustness of learning, RL can still be slow and may not converge to an optimal solution.

In a different project, genetic programming is used to build routines for finding a left and a right hand on the image of a human for the augmented reality system ALIVE [32, 33]. Following Koza's genetic programming algorithm [38], the author defines a set of terminals (centroid of the silhouette, bounding box, etc.) and primitive functions (point operators, edge

detectors). It is assumed that early vision stage outputs are available and noise-free. Evolved routines achieved mean detection accuracy of $77 \pm 21\%$ compared to human performance. The questions of how a set of elemental operations should be formed, whether it can be learned or predefined and whether it should be fixed or expanded are left unaddressed by the researchers. In all cases elemental operations are specific to the task.

Although attention played a major role in the original formulation of visual routines, very few projects implemented it. For example, in Jeeves and VR Framework it is represented by registers for indexing operation and an inhibition-return map for visual search. Majority of works considered here interpret attention as a selection of region of interest (e.g. Sonja and AV-Shell) to minimize the amount of processing.

Rao [65] explicitly mentions attention and attention state in his work. Attention state contains a current object of interest, its attributes, its local context and also a history of previous attention states. However, functionally his system is not different from the other systems discussed earlier. In his work focus of attention is a point in the image and saccades simply change the coordinates of this point. There is no fovea or explicitly defined region of interest around the focus of attention and the size of the local focus is limited by the size of the largest Gaussian filter applied at the early visual processing stage.

In conclusion, the implementations of visual routines discussed in this section provide a computational argument that task-specific routines are efficient and even a small amount of strategically extracted visual information is sufficient to perform complex visual tasks. The top-down nature of control for particular tasks and specificity of visual representations also has supporting evidence from psychophysics. As an example, experiments by Hayhoe [30] demonstrate that visual system represents information necessary for the immediate visual tasks. From a neurophysiological point of view Roelfsema [67] outlines how components of visual routines (base representation, elemental operators, etc.) could be represented by neurons in the cerebral cortex.

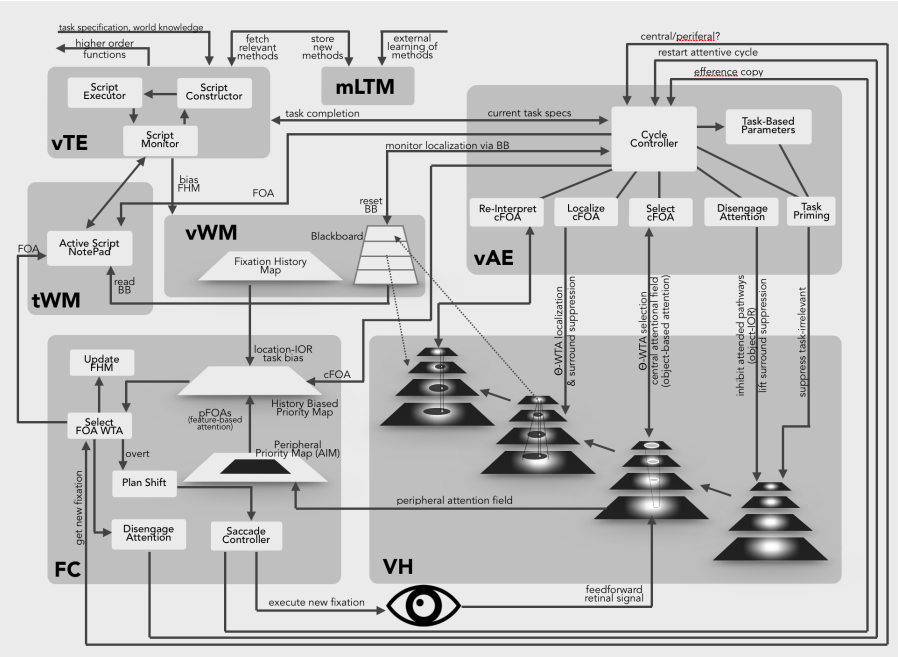
The previous attempts at implementing visual routines exposed some practical issues not anticipated in the theoretical formulation. For example, it is not clear what determines the

size of the attentional 'spotlight' and other parameters of visual attention. Tuning a system parameters based on an arbitrary task is still an open research problem, as of now suitable values for parameters are usually hard-coded for each case.

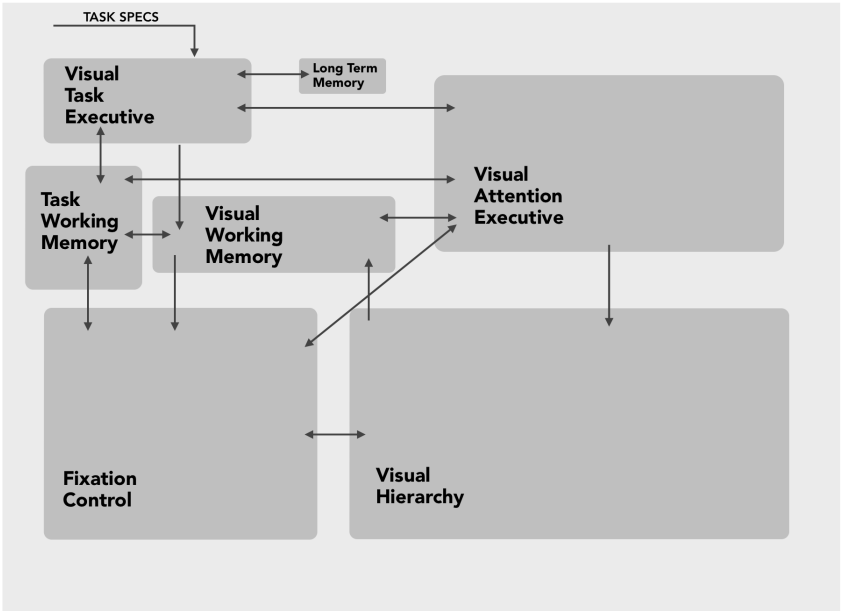
However, even despite the gaps in the theory of visual routines it has been successfully applied in multiple domains, such as visual search in real and simulated domains, autonomous driving, playing computer games, etc., and deserves further investigation.

2.3 Cognitive Programs and Selective Tuning

The mainstream models of visual attention can be summarized as the selection of region of interest, maintaining a list of locations where focus of attention should move next, and a list of already attended locations to prevent cyclic behavior. An important assumption is that a complete and constant information about the scene is a product of a single feed-forward pass within the vision system. In Ullman's paper on visual routines and their multiple implementations a base representation contains all the knowledge about the scene. This view is also common in the cognitive research literature, both in architectures designed for simulated environments (ACT-R [53], EPIC [37], Homer [90], Soar [91]) and in systems, which work with real environments (RCS [2], Polyscheme [12]). However, advances in the neurophysiology of vision since the 1980s established that the visual system is likely more complex. For example, an obvious problem with the idea of completeness of the base representation is a non-uniform distribution of receptors in the human retina. Consequently, the acuity of vision is greater in the center field of vision (2° of visual field) compared to the periphery. The hierarchical organization of human vision also does not agree with the idea of immediate availability and constant nature of base representations, since computation of some features may be affected by both feedback and lateral connections within the hierarchy. These are just a few examples demonstrating a need for a more complex model of visual attention that could accommodate new findings about the human visual system. A more detailed discussion of this subject is beyond the scope of this thesis and can be found in [39].



(a) Detailed diagram of Cognitive Programs



(b) High level diagram showing connections between major components of CPs

Figure 3: Cognitive Programs (images adopted from [85])

The concept of Cognitive Programs (CPs) [85] is an updated version of visual routines as a theory of executive controller for Selective Tuning (ST) [84] model of attention. A detailed diagram of CPs is shown in Figure 3a and high level connections between major parts of the system are outlined in Figure 3b.

ST is represented by a Visual Hierarchy (VH) box in the diagram. VH models a set of neurons organized as a hierarchy of layers, where connections exist between neurons within the same layer and layers above and below. Selection of the required feature or location within each level of the pyramid is done via the Θ-WTA [1] (winner-takes-all) algorithm. In the diagram several consecutive stages of processing are shown: a) hierarchy, primed for the target to appear in the center of the image (bottom plane), b) feedforward pass, c) recurrent top-down localization suppressing close neighborhood of the target location, and d) feedforward pass with suppressed units. Not all stages are necessary for every visual task, for example discrimination and categorization would stop after the feedforward pass b). Straight lines within the pyramid in the diagram show the current focus of attention (FOA). All information in the current FOA, including selected features, fixation location and other parameters, is called an attentional sample. The attentional sample is only roughly analogical to the common notion of 'spotlight' of attention, since its functions go beyond selection of region of interest. When attention is shifted to a new location, a new attentional sample may be generated and saved in visual working memory blackboard (vWM in the diagram) to be used by other processes. The shape and the size of the attentional sample depends on the parameter Θ in WTA. This parameter can be modified depending on the task. Since ST hierarchy contains several types of neurons, it allows priming not only for spatial locations where the target is likely to appear, but also for particular features (color, motion, etc.).

The rest of the diagram contains the mechanisms for tuning and controlling the execution of ST. Fixation Control (FC), as its name suggests, is responsible for gaze change. It takes into account saliency of the peripheral visual field ($> 10^\circ$ in the early layers) represented

[1] Classic WTA ensures that a neuron with the highest activation within a layer stays active, while all others are suppressed. Θ-WTA relaxes this requirement, so that several neurons with firing rates within Θ of one another can be active at the same time.

by the Peripheral Priority Map (PPM), which is built using a bottom-up saliency algorithm called Attention based on Information Maximization (AIM) [10]. History Biased Priority Map (HBPM) combines the most salient items of the PPM with the focus of attention from the central visual field (cFOA), which is the result of processing by the visual hierarchy. The next fixation can be selected either from the items in the central visual field (and would not require movement of the eye) or from the peripheral fixation items depending on the task.

Setting the parameters of the Visual Hierarchy and gaze control are done via cognitive programs, which are composed of various operations or other CPs and can be of two types - *methods* and *scripts*. This is different from the distinction made in [86] between universal (applied to any part of the image) and regular (applied to the result of universal or another regular routine). In CPs methods are blueprints of the operations, specifying parameters and general flow of computation. Scripts are methods with particular values for parameters in place and ready for execution. For example, a method for visual search would require a target to be specified. Specific operations and parameters would depend solely on parameters of visual and control system to be tuned.

Also note that visual routines were originally described as chains of functions, where the input of one function was the output of the previous function in the chain. Consequently, no mechanism for storing intermediate results was provided and elemental operations do not include I/O functions. This problem was acknowledged by the successive implementations of visual routines. All of them allocated registers for saving the indexed locations or intermediate results of computations, although none of them related this to a concept of visual memory.

Cognitive Programs employ several types of memory to store and manage cognitive programs: long term memory, visual working memory and task working memory. Generic methods are stored in the Long Term Memory for Methods (mLTM). At this point methods are predefined or learned by the system external to CPs. Visual Working Memory (vWM) contains the history of previous fixations - Fixation History Map (FHM). It is primarily used to bias against revisiting previously seen locations but can also be overridden by the demands of the task. Blackboard (BB) is also a part of vWM, which makes the attentional sample accessible

by all other components. Task Working Memory (tWM) saves information about the scripts in progress in the Active Script NotePad. It has access to the contents of current focus of attention (FOA) and blackboard (BB) with previous attentional samples from visual working memory.

The Visual Attention Executive (vAE) includes a Cycle Controller, which initiates and terminates each stage of Selective Tuning. It runs in the background until the visual Task Executive (vTE) sends a command that a task is finished.

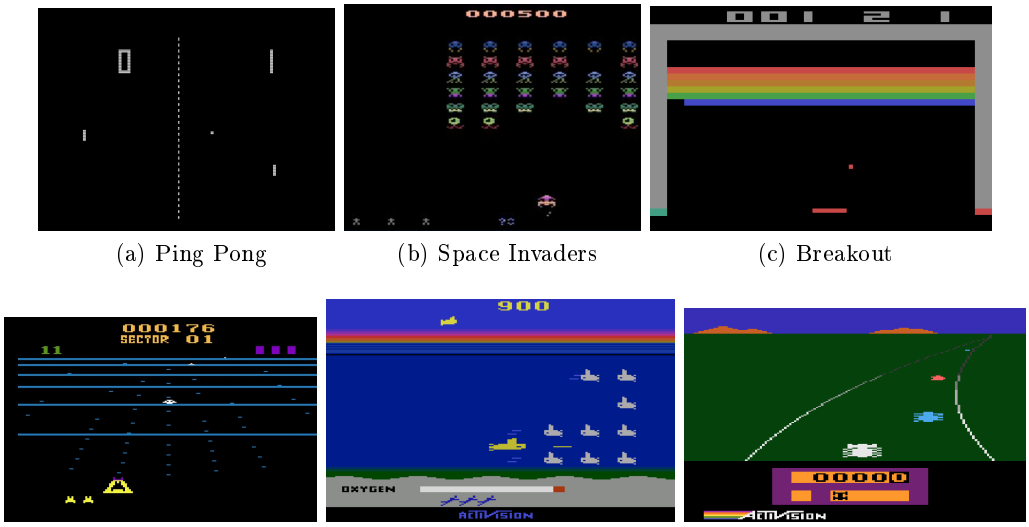
The Visual Task Executive (vTE) receives a task description, selects appropriate methods, tunes them into scripts and runs them using the data stored in Active Script NotePad.

The Selective Tuning model has been in development for over two decades. Parts of it, Visual Hierarchy and Fixation Control have been implemented using the TarzaNN neural network simulator. Most recent results can be found in [92, 72]. Components required for retrieving, tuning and controlling the execution of Cognitive Programs have been hypothesized. This thesis is a first attempt at testing components of the new structure. The test domain is the task of playing a video game.

2.4 Game AI for platform games

Some of the earliest examples of game AI can be found in the single-player games like Pac-Man (1980) [47]. The first AI agents were designed as a set of hard-coded rules. Eventually many new genres were introduced making video games a challenging environment for research in general AI. Currently, game AI research focuses on both designing artificial opponents to the human players and imitating the human style of playing using classic AI tools (path-planning, finite state machines) as well as machine learning techniques.

Since the mid-2000s various game AI challenges became a noticeable part of the AI research, many of them run by universities and as part of conferences. For example, the IEEE Conference on Computational Intelligence and Games hosted six game AI competitions in 2015. Traditionally, the game AI research focused on a particular game or genre, and only recently initiatives like the General Video Game AI Competition began exploring the problem



(a) Ping Pong (b) Space Invaders (c) Breakout

(d) Beamrider (e) Seaquest (f) Enduro

Figure 4: Screenshots of the popular Atari 2600 games (source: atarimania.com)

of creating controllers capable of playing multiple games. Performance-wise the most successful attempt to date was a deep neural network trained via reinforcement learning to play 61 Atari games without parameter adjustment [50]. The system could play more than half of these games better than the human expert players. Although it is rather impressive that a non-trivial sequence of commands for games representing a wide range of genres can be learned from the raw screenshots, this model provides little insight about the cognitive processes that led to this performance. Even though it is possible to visualize hidden layers of the network and associate particular parts with certain games, it does not help to understand why Pinball is the easiest game to play (4500% better than human performance) and Montezuma's Revenge is the hardest (cannot be played at all by the model).

Overall, machine learning based solutions for a single game are almost universally outperformed by agents based on heuristics (Table 3), classic AI (Finite State Automata, behavior trees, A*) and combinations of the two, especially when complete noise-free information about the environment is given. For example, in the MarioAI competition the organizers were forced to put restrictions on the time and amount of information available from the simulator after

A*-based agents achieved the highest scores for several years in a row. In the Angry Birds AI competition the runner-up algorithm based on Bayesian Ensemble Regression scored only half of the points that the first place algorithm earned. In Geometry Friends, A* and Q-Learning algorithm placed second with 1/4 of the leader's points. In GVG-AI the runner-up algorithm based on a combination of offline and online learning techniques scored only 8 points less than a leader.

A large study on general Atari game playing (featuring 61 games) by Hausknecht *et al.* [27] compares the performances of various neuroevolution algorithms depending on what kind of game state representation is provided: raw pixel (downsampled screenshots), object (by template matching) or noise-screen (randomly selected points). Noise-screen representation was used as a baseline to determine the learning capabilities of the algorithms. Raw pixel representation did not lead to good performance and a large amount of data was required for training. Object representation significantly outperformed other representations for all learning algorithms, but presented a few issues: templates had to be extracted beforehand and neural networks had to be configured for a particular number of objects. As a result, never before seen objects were assigned a label of the most similarly looking known class.

There is a general agreement that interaction with computer games is similar to the interaction with the physical world, albeit in a simplified and controlled environment. However, the problem of uncertain and noisy perceptual information is still rarely addressed. Most AI research is conducted on simulators of games. One notable exception is the Angry Birds AI, where locations of various objects on the screen are found using color segmentation from the screenshots of the Angry Birds running in a browser window. However, both vision and physics modules are provided as part of the competition software and participants are not expected to improve them. Sometimes noise is artificially added to the simulated sensor readings to make it more realistic, like in the case of Simulated Car Racing Championship. In all other cases the information about the world is complete and correct.

Game AI for platform games (platformers) deserves a more detailed discussion as the most relevant to our work, particularly, Super Mario Bros. as one of the most representative games of

Competition name	Task/Winner
The General Video Game AI Competition 2005-present	**Task:** Playing multiple games of various genres (puzzle, strategy and 2D platform games) **Winner:** An agent using Open Loop Expectimax Tree Search [60]
Simulated Car Racing Championship 2007-present	**Task:** Design the best autonomous car controller to win a car racing game **Winner:** GRN Driver agent using a Gene Regulatory Network evolved with Genetic Programming to control car steering and throttle [75]
The 2K Bot Prize 2008-present	**Task:** Imitate human player in first-person shooter game Unreal Tournament 2004 **Winner:** MirrorBot agent, which switches between default behavior (graph-based planning) and mirroring behavior (recording and playing back actions of another player) [62]
AI Challenge 2009-2011	**Task:** Write an algorithm to play a chosen game against an opponent **Winner:** Agent xathis for game Ants won in 2011 using a combination of classic AI planning and heuristics
Mario AI Championship 2009-2013 Platformer AI Competition 2013-present	**Task:** Playing Infinite Mario Bros. **Winner:** REALM agent with hybrid architecture, where A* is used to find the best actions evolved by genetic programming [7]
IEEE CIG StarCraft AI Competition 2010-present	**Task:** Compete with other bots 1 vs 1 in a full game of StarCraft in the round-robin format **Winner:** ZZZBot acts according to a set of predefined rules
AIIDE StarCraft AI Competition 2010-present	**Task:** Compete with other bots 1 vs 1 in a full game of StarCraft in the round-robin format **Winner:** AIUR 2.2 bot selects a random behavior from a predefined list and records the runtime statistics to update the probability distribution for future behavior selection [55]
Ms Pac-Man vs Ghosts League 2011-2014	**Task:** Develop AI controllers for the classical arcade game Ms Pac-Man **Winner:** ICEP_IDDFS based on iterative deepening depth-first search [59]
Angry Birds AI Challenge 2012-present	**Task:** Build an AI player that can play new game levels as good or better than the best human players **Winner:** DataLab (2015 winner) with agent selecting a strategy from a list of predefined strategies [66]
Geometry Friends AI Competition 2013-present	**Task:** Play as one or both characters in the puzzle game Geometry Friends. **Winner:** CIBot using Monte-Carlo Tree Search with Directed Graph Representation[63]
Fighting Game AI Competition 2013-present	**Task:** Build controller for Java based fighting game FightingICE **Winner:** Agent called Machete following predefined rules
2048 Controller Competition 2015	**Task:** Learn an evaluation function for 2048 - single-player, non-deterministic, online puzzle game **Winner:**Agent IeorIITB2048 using cross-entropy method [35]

Table 3: Chronological list of popular game AI competitions. The most recent winning algorithms using machine learning methods are highlighted.

the genre. Majority of the research on Super Mario comes from the MarioAI Championship, which ran from 2009 to 2012 [82]. Its most valuable outcome was the reinterpretation of platform game playing as a planning problem. In platform games score depends both on time it takes to clear a level and bonuses collected in the process. Thus, playing the game can be viewed as finding an optimal path through the environment. Classic A* search [58] then can be applied to find the best solution. A* is a graph search algorithm, which finds the path with the lowest cost between the start node and one or many goal nodes. Using predefined heuristics the cost of each path is estimated and a decision is made whether to add the node to the final sequence or not.

In the case of Mario Bros. each node is a current world-state defined by the position, speed and state of Mario, his enemies, bonus items and immobile objects. Possible next states are determined by the avatar's actions - jumping, moving to the left or right, ducking or firing, and can be computed precisely by accessing the physics engine to simulate the next step. The scope of the available information and hence the number of steps to plan ahead is limited by what is visible on the screen. The 40 ms time limit before the game is updated in most cases is not enough to find an optimal solution.

The algorithms submitted for the Mario AI championship can be split into three groups: classic search algorithms (A*), rule-based and learning-based (neural nets, genetic programming and imitation learning). Top results were achieved by A*-based or hand-coded rule-based algorithms. All solutions based on various learning algorithms performed significantly worse, in fact none of them surpassed the score of the baseline ForwardJumpingAgent supplied as part of the competition software.

As has been mentioned, additional restrictions such as limited scope and dead ends were introduced to make the use of A* impractical. Subsequently, none of the submitted controllers were able to clear all levels without losing and those based purely on A* were not the front runners anymore [36]. For instance, the winning hybrid agent called REALM [7] uses genetic programming to evolve a set of rules from a smaller initial set of hard-coded rules. Then at each point it evaluates the current state of the game and picks an appropriate rule from the

set or a default action (move forward). After that it uses A* to find an optimal path through the environment.

Overall, only A*-based agents were able to play Super Mario Bros. perfectly, given that the input to the algorithm was noise-free and there was enough time for computation, while all machine learning controllers were significantly worse. The purely rule-based agent for the Super Mario performed at $\sim 75\%$ of the top score, but was more than 100 times faster. This also holds for the games of other genres (see Table 3), where machine learning is outperformed by the classic AI and heuristic based algorithms for playing a single game.

2.5 Summary

In this chapter we placed Cognitive Programs within a context of other relevant areas of research, namely the cognitive architecture design and top-down control of visual attention. We showed that the problem of connecting sensory data to higher order cognitive functions is still largely unresolved. There are architectures like RCS, which excel at combining information from multiple sensors and resolving multiple issues caused by the noise and dynamically changing environments, but they are designed to perform very particular tasks. On the other hand, established architectures like Soar and ACT-R apply their high order reasoning only in simulated environments.

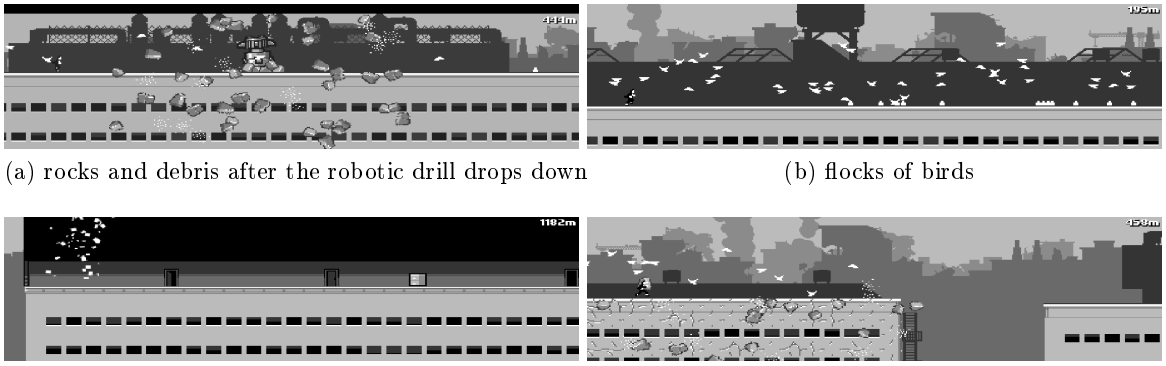
Although elements of top-down control like region of interest selection or feature selection are very common in practical applications, there are few models that try to solve this problem in general. One of them, called Visual Routines, we described in more detail. Ullman's idea to represent vision as modular process led to a string of successful implementations, but also revealed practical issues not anticipated in the theoretical formulation. CPs follow a similar approach in design of executive for visual attention and utilizes a more modern understanding of vision and computational resources not available in the 1980s, when the first paper on visual routines was published.

In order to design a part of CPs which control the visual task of playing a computer game, we also reviewed current approaches to designing AI agents for games similar to ours.

3 Problem Statement and Implementation

3.1 Problem Statement

The main goal of this thesis is to test aspects of the Cognitive Programs concept by applying it to the task of playing a computer game using only visual input. Since most of the control system has been developed only theoretically, we are hoping that our attempt would provide a practical justification for the system as well as reveal possible design flaws to facilitate further development.



(a) rocks and debris after the robotic drill drops down

(b) flocks of birds

(c) shards of glass

(d) collapsing building

Figure 5: Screenshots from Canabalt demonstrating various visual distractions typical for the game

We propose to use two modern browser games for a real-time system of visual attention based on the theory of Cognitive Programs. Unlike similar projects from the past that also applied the concept of visual routines to computer games, our algorithm has no access to the game engine, works on a full-resolution screenshots and does all processing in real-time. Additionally computer graphics has evolved since the 1990s and as a result even simple browser games can present a challenge for the current state-of-the-art vision algorithms.

Although not as complex as real images, the visual environment of modern computer games is nontrivial, dynamic and requires more sophisticated computer vision techniques than template matching and color segmentation. Besides being graphically interesting, many popular

modern games have a minimalist gameplay. This avoids the problem that some of the past projects experienced, where the game was too complex, had unclear objectives and too many items/enemies.

Canabalt (2009) and its clone Robot Unicorn Attack (2010) are the first endless scrollers featuring an infinite procedurally generated environment. The objective of both games is to run as far as possible, jumping over the gaps and avoiding hitting objects, which emphasizes reaction speed and attention over planning. Since the games run in a browser window, the algorithm playing it has the same amount of information as the human player would have (the games also provide sound clues to warn about obstacles ahead, but they are ignored). Lastly, the game cannot be stopped or slowed down, meaning that all decision making has to be done in real-time.

3.2 Modifications to Cognitive Programs

The task of playing the games can be formulated as follows: for each frame decide whether to press/release the button given the world state (edges of the buildings, locations and types of objects and motion properties).

Figure 6 shows the parts of Cognitive Programs in our implementation which differ from the original formulation. Both Visual Hierarchy and Fixation Control exist as standalone applications, but they do not run in real-time. For our project both components were reimplemented on GPU from scratch with some changes and optimizations. For instance, because of the time constraint, only the bottom level of the pyramid is used for computations. The tasks originally performed in a serial manner are done in parallel on GPU, e.g. recognition. The localization step is done by the mean-shift algorithm [23]. Spatial priming is performed when searching for the character, since its approximate location is known. Another instance of spatial priming is for the objects, in this case we use the fact that objects are located on top of the platform.

The primary role of FC is to plan the next move while taking into account the salient objects in the peripheral priority map (PPM) computed using the bottom-up saliency algorithm AIM

[10]. However, in our case the next fixation is always determined by the gameplay requirements (e.g. track the obstacle or follow the edge of the rooftop). We originally ran tests to determine if early detection of obstacles via PPM would improve results, but found that moving gaze horizontally along the current platform perform better, which was also suggested by eye-tracking data from human subjects. This part of the code is not used in the final version of the algorithm.

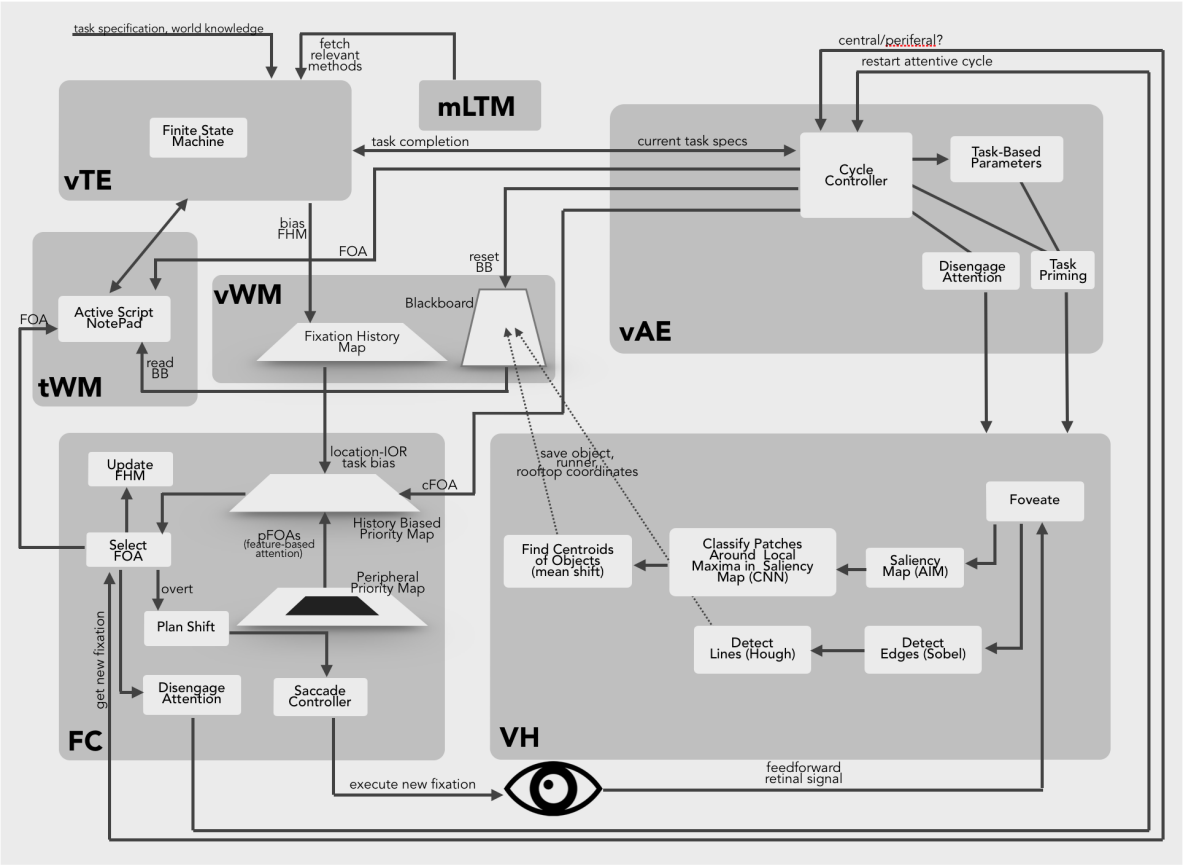


Figure 6: Cognitive Programs diagram with modifications

The visual attention executive (vAE) now mainly serves for task priming and controlling the contents of the visual working memory (vWM). The vWM contains a Fixation History Map, which saves fixations in the previous frames and also a Blackboard (BB), where locations of all objects and lines in the current frame are saved and made available for all other elements

in the diagram (they represent an attentional sample for the current frame). Task Working Memory (tWM) reads locations of the objects and lines from the Blackboard and stores them in the Active Script NotePad.

The Visual Task Executive (vTE) is responsible for coordinating all these modules. In our case the cognitive programs are hard-coded and there is no need for composing scripts and tuning them. Therefore we did not implement the Script Constructor. The Script Monitor has access to Active Script NotePad, which contains all information relevant for the execution of the scripts (e.g. coordinates of objects in the current and several past frames, variables and timer for a button press). The vTE also calls external functions to measure the speed of the game, compute jump trajectory and determine the duration of button press.

Keyboard input is required to control the character on the screen, however, the original diagram of CPs in [85] does not specify how it interacts with motor functions. Therefore, we assume that keyboard calls can be done through vTE.

3.3 Workflow of the Model

In this section we describe technical details of implementing concepts of Cognitive Programs. In our implementation the vTE acts as a rule-based game AI. In order to develop a set of rules we analyzed recordings of the eye movements of several human players while they were playing Canabalt and Robot Unicorn Attack. In section 3.3.1 we describe the pilot study and give qualitative analysis of the results. In the following sections we discuss how these rules are implemented for the two games and also outline steps of the vision processing and algorithms applied.

3.3.1 Eye-tracking pilot study

We conducted a small experiment where we asked 3 subjects[2] to play several sessions of Canabalt while tracking their eye movements with the Pupil eye-tracker [3]. Some examples are shown in the Figure 7. Our analysis of the data was limited to finding patterns in the eye movements, which would be helpful in developing the game playing logic for the game.

Here are the main observations :

- all eye-movements were made horizontally along the tops of the rooftops, rarely looking below or above;
- since the game scrolls automatically, there is no need to control the character while it is running on the top of the platform. Consequently, players occasionally look back at the character and immediately do a horizontal scan until an obstacle (gap, crate or robotic drill) is found (keyframes 1-3 in Figure 7);
- once an obstacle is seen, it is tracked until a decision to jump is made (keyframes 3-5 and 9-10 in Figure 7 show tracking of the gap between the platforms, keyframes 7-8 show tracking of the robotic drill);
- finally, when the decision to jump is made, the gaze is moved to the right towards the next closest obstacle (keyframes 5-6 in Figure 7).

In Canabalt rarely more than one obstacle is placed on a single platform, therefore all decisions about the obstacles are made in FIFO order one at a time. For example, if there is a gap followed by a box on the next rooftop, first the gap will be tracked and only once the jump is made, the gaze would move on to the next rooftop and only then to the box on top of it.

We did not continue with a full study of eye-motion with naive participants. The pilot study was conducted in order to find out if any useful strategy for playing the game can be extracted from the eye-tracking data.

[2] Our subjects were one female (the author) and two males, all members of Tsotsos lab. The author was the only expert player of the game, both other subjects played the game for the first time. Participants were not paid for their time.

[3] `https://pupil-labs.com/pupil/`

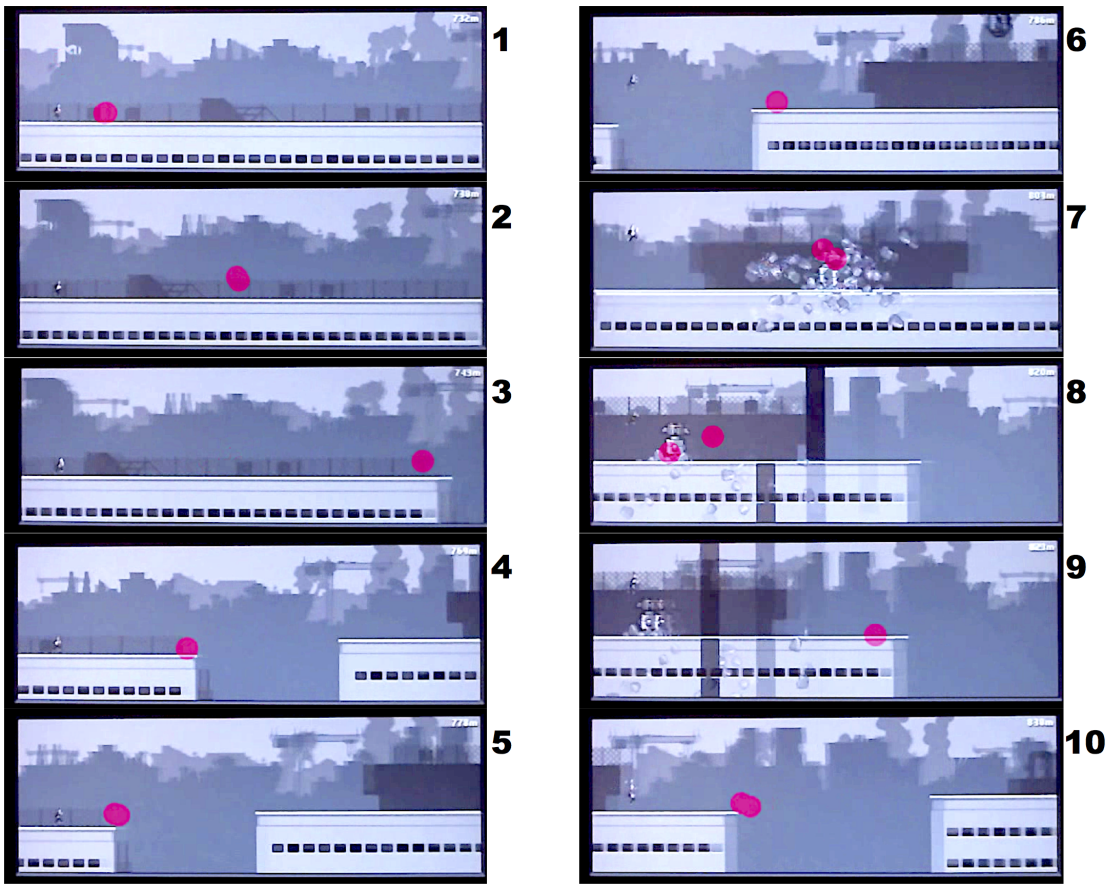


Figure 7: Recording of eye movements of the author playing a session of Canabalt game. Figure shows keyframes from approx. 100 frame sequence with fixations shown as red dots. Here we can observe searching for the next obstacle (1-3, 5-6, 8-10) and tracking an obstacle (3-5, 7-8). Note that all fixations are distributed horizontally along the top of the platform.

3.3.2 System setup

The code for this thesis was written mostly in C. Small parts were implemented in other languages: OpenCL 1.2 was used for all visual processing and OpenGL 4.3 with GLSL 4.2 for visualization. No multi-threading was used. In addition MATLAB scripts were written for training convolutional networks, gathering and analyzing various game-related data (game over screenshots, jump trajectories, image patches with different objects in the game, etc.).

All experiments were conducted on a desktop with the following specifications:

- CPU: Intel Core i7-3820 @ 3.60GHz with 8 cores
- RAM: 16 GB DDR3
- GPU: two AMD FirePro W7000 (Pitcairn XT GL) video cards with 4GB DDR5 RAM
- OS: Ubuntu 12.04.5 LTS

3.3.3 Canabalt

We applied the observations made from human player data to the design of the Cognitive Programs for playing Canabalt. Figure 9 shows a diagram with a high level description of methods (e.g. find a running man) and what actions within the CPs framework are required to execute them. Finally, each path within a diagram also represents a method in the CPs terminology. In our implementation the vTE acts as a rule-based game AI.

A high level description of the strategy to play the game amounts to a few simple rules: look at the character, scan to the right until the first obstacle is found, track the obstacle until a jump can be safely made and immediately start looking for the next obstacle to the right.

At the low level, when the game is started, all parameters of the system are reset to the default values, the visual hierarchy is primed to look for the character in the left half of the screen, and the gaze location is moved to the left. When the first screenshot of the browser window is taken and loaded into the visual system, it is foveated and filtered to find edges and salient regions. Randomly selected points within salient regions are passed through a convolutional neural network (CNN) [43] for recognition. Finally, localization of recognized objects within fovea is done via mean-shift [23]. All detected edges and objects are saved in the visual working memory (vWM). Active Script NotePad in the task working memory (tWM) contains all data related to the current task, i.e. gaze location, platform coordinates, current and previous speed estimates, parameters of the button press, jump trajectory, distances to the objects, etc.

The vTE controls the execution of the task based on the rules and the information within the visual working memory and task working memory. For example, to check if the character

(a) Image pyramid

(b) Foveated screenshot

Figure 8

is found in the image, the vTE examines the contents of the Active Script NotePad. If the character is not found and its previous location is not within the current fovea, it means that the system is currently tracking an obstacle or looking for the end of a rooftop. In this case the location of the character is assumed to be unchanged and the new frame is loaded. Otherwise, if the runner is found within the fovea and is not performing a jump ('is runner on the rooftop?'), then we check for obstacle locations recorded in tWM. If nothing is found within the fovea and the rooftop extends beyond it, then the gaze is moved to the right along the current rooftop and the next frame is loaded. When an obstacle is sufficiently close to the character (the distance depends on the current framerate and speed), a decision is made to jump. The vTE calls an external function to compute the trajectory of the jump and the duration of the button press and then sets the timer and a flag for the button pressed in tWM to true and loads the new frame.

Below we describe the processing and algorithms required to implement the elements in the diagram.

START. When the new game is started, all variables in the game are reset to defaults. Since we expect the character to always be in the left 1/3 of the screen, we set the gaze location

at $(\frac{1}{2}h, \frac{1}{3}w)$, where w and h are width and height of the frame. This way the vision system will have a chance to recognize and localize a character and most of the scene will also be visible.

LOAD NEW FRAME. A screenshot of the rectangular area within the browser window is taken via Xlib, then a resulting image buffer is converted from XImage to image2d_t format (unsigned char array) and resized from 320×920 to 256×1024 pixels to improve GPU processing.

Next, the image is foveated using the current gaze location (Figure 8b). Our implementation of foveation process is based on the BlurredMipmapDemo from Matlab Psychophysics Toolbox [8]. First, we build a Gaussian pyramid (without scaling), combine levels of the pyramid so that the fovea contains pixels from the first (not blurred) level, and copy rest of the pixels from the different levels of the pyramid depending on the distance from the center.

Foveation is a computationally expensive operation, therefore it is done on GPU. Since OpenCL 1.2 does not have a built-in function to create image pyramids, we implemented it ourselves. For performance reasons all levels of the pyramid are computed in a single buffer (Figure 8a). Instead of building a Gaussian pyramid explicitly by filtering (which requires two passes per level), we can exploit the built-in image sampler. The result is equivalent to applying a box filter. Linear interpolation is implemented in the hardware on most modern video cards and is extremely fast. Using this technique the pyramid on GPU can be created by repeatedly resizing the original image with linear interpolation sampler and then combining blurred versions in another kernel call. In our implementation the pyramid has 6 levels and requires 7 kernel calls.

Following [77] the fovea diameter is set to 2°. If the distance from the player to the monitor is 57cm (22.44 in), pixel density (ppi) of the monitor is given by

$$\text{PPI} = \frac{d_p}{d_i} = \frac{\sqrt{w_p^2 + h_p^2}}{d_i} = \frac{\sqrt{1920^2 + 1080^2}}{23} = 95.7786$$

where w_p add h_p are the dimensions of the screen in pixels and d_i is the diagonal of the screen in inches. Using simple trigonometry the radius of the fovea in pixels can be calculated as

Code Listing 1

```
//OpenCL pseudocode for resize kernel
//one thread per pixel in output image
//dimensions of output image are half of the input
//kernel runs for each layer in the pyramid
x_in = x_out/width_out

//interpolation step
y_in = y_out/height_out pixel = image[x_in, y_in]
image[x_out, y_out] = pixel
```

Code Listing 2

```
//foveate kernel (runs once all layers are computed)
//one thread per pixel in foveated image
//gaze_pos is passed as an argument
//lod (level of detail) is the level of pyramid
//where pixel should be copied from

//0.02665 = 1/37.5172, where 37.51 is the radius of the fovea
lod = max(0, log2(distance(gaze_pos, (x, y))*0.02665))

//if lod is a fraction, then interpolation is needed
level1 = max(0, floor(lod))
level2 = min(ceil(lod), 5))
l1 = 1 << level1 l2 = 1 << level2

//pyramid can be 2D or 3D image if the video card supports it
m = lod - level1 pix1 = image[l1, x, y]

//interpolate between two levels
pix2 = image[l2, x, y] fov_image[x, y] = pix1(1 - m) + pix2*m
```

$$d_s \cdot \tan(1) \cdot \mathrm{PPI} = 22.44 \cdot 0.017 \cdot 95.77 = 37.51$$

where d_s is the distance from the screen.

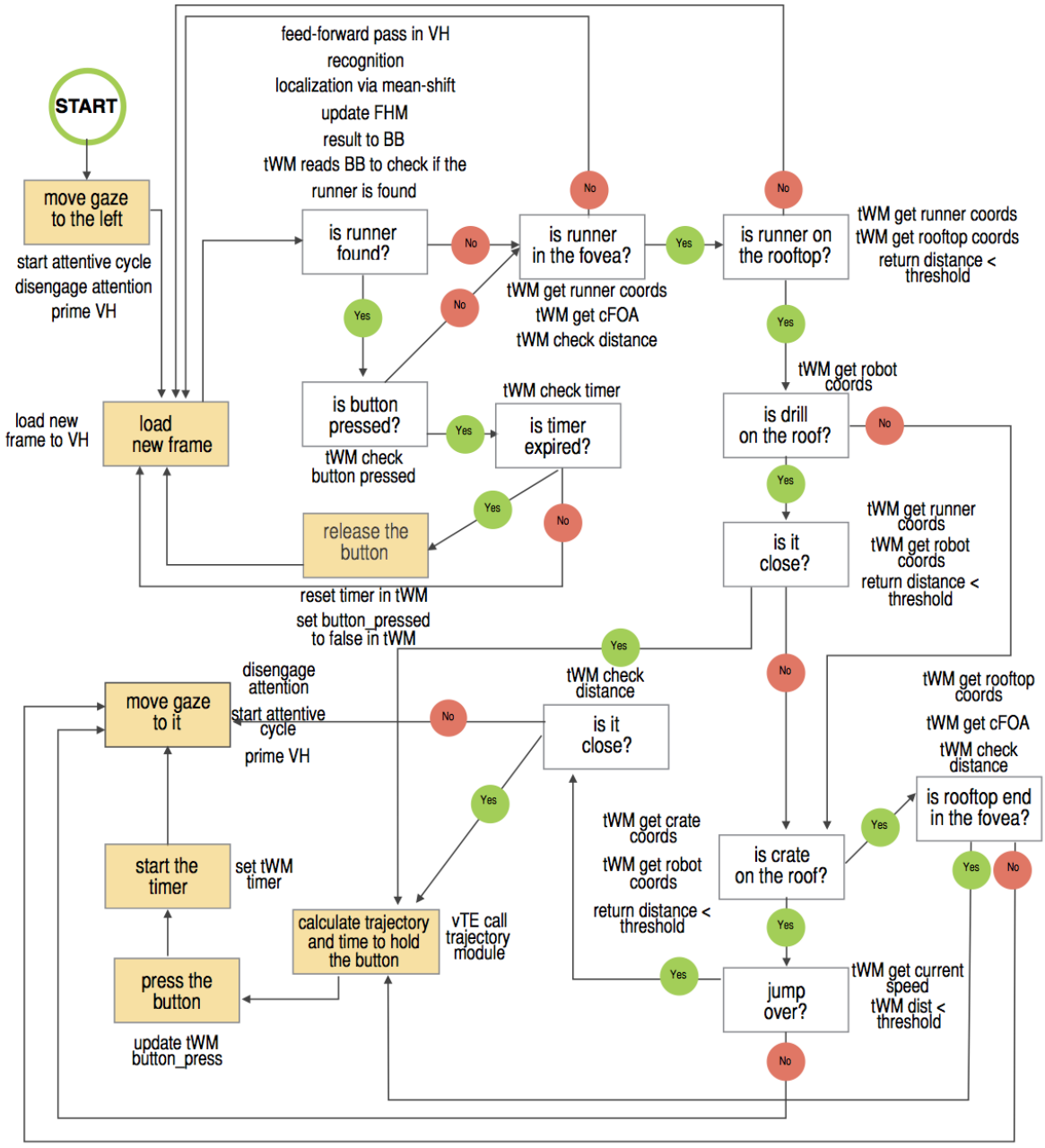


Figure 9: Diagram of Cognitive Programs for the game playing task. The game playing logic is implemented as a decision tree: white boxes show methods and yellow boxes show actions (pressing a button, changing gaze, etc.). Small text next to the white boxes describes operations within CPs framework required to implement a given method.

VISUAL PROCESSING. The feed-forward pass in VH computes line segments and recognizes salient objects. Localization is simulated by the mean-shift algorithm [23].

Edges are detected by filtering the foveated image with 3×3 Sobel operator. To find line segments we use the fast Hough transform optimized for GPU [87]. In the naive implementation the kernel is started with one thread per pixel. If the pixel value is non-zero, the vote is placed in the Hough space for each possible angle. Since edge pixels normally take up less than 5% of the image, very few threads will be performing a large number of operations to the matrix serially. Thus to better utilize capabilities of the GPU two kernels are needed: one builds an array of the coordinates of edge pixels and the second populates a matrix of votes in the Hough space. The trick with two kernels significantly improves performance by efficiently distributing the work between threads for the second more computationally expensive kernel. Since all lines in the game are either vertical or horizontal, we further reduce the amount of work by computing the Hough transform for few degrees around $0°$ and $90°$. The line segment endpoints are computed on CPU. As a result, for each frame at most 10 lines with lengths of 100 or more pixels are detected. If the total length of gaps is more than 15% of the line length, it is discarded.

In order to find regions of interest in the image we use a bottom-up saliency algorithm AIM. This algorithm assigns higher saliency to image patches that are unexpected given their context. Computation of saliency map consists of several steps: first, a set of independent basis functions is learned from multiple examples. Next, a patch around each pixel in the image is multiplied with each basis function resulting in array of responses. These responses yield a distribution of values for each coefficient in a form of histogram. A probability of each value then can be calculated. A product of all individual probabilities corresponding to a particular patch represents the joint likelihood, which is translated into Shannon's measure of Self-information by $-\log(p(x))$. Finally, the saliency map is normalized to range$[0, 1]$.

We ported the original Matlab code for AIM to OpenCL. It is split into 5 kernels calls:

1. `project_into_basis` - image is convolved with each filter from the basis;
2. `min_max_reduce` - compute min and max value among projection results for all features;

Code Listing 3

```
//OpenCL pseudocode for Hough algorithm
//collects coordinates for all non-zero pixels in the image
//each group collects coordinates in local memory and
//then copies them to the global array
__local int coords[256]
if image[x,y] > 0  {
    coord = (x << 16) + y
    //increment index atomically to avoid threads
    //overwriting each others results
    idx_ = atomic_inc(&idx)
    coords[idx_] = coord
}
//find index in the global array for this workgroup
start = atomic_add(global_offset, idx)
//write to global array
output[start+thread_id] = coords[thread_id]

//hough_count (builds matrix of votes in Hough space)
y = coords[global_id] & 0x0000FFFF
x = (coords[global_id] >> 16) & 0x0000FFFF
for theta=0:180 {
    rho = y*sin(theta) + x*cos(theta)
    rho_indx = round(rho - firstRho)
    //atomic increment the output
    //at this theta and rho
    atomic_inc(output[theta, rho_index])
}
```

3. `histogram` - rescale the projection result for all features to be between $[0, 1]$ using min/max values computed in the previous step and build 256-bin histogram of the rescaled values;
4. `sum_partial_histograms` - for performance reasons histogram is computed in two stages;
5. `compute_saliency` - final step of the computation which outputs a single saliency image.

Next we select at most 100 salient points[4] in AIM saliency map on CPU as follows:

- starting from the top-left corner find a local maxima above the threshold (approx. half of the max saliency value);

[4]Since recognition is performed on the GPU, the structure holding local maxima cannot be dynamically allocated. Therefore, we gathered statistic over several thousand frames from the game and determined that majority of frames contain only a few objects, however frames with flocks of birds can have up to 50. In order to ensure that every local maxima is processed, we doubled that number.

Code Listing 4

```
//OpenCL pseudocode for AIM
//project_into_basis (one thread per pixel)
for f=0:NUM_AIM_FEATURES {
    offset_x = -AIM_PATCH_RADIUS:AIM_PATCH_RADIUS    {
    offset_y = -AIM_PATCH_RADIUS:AIM_PATCH_RADIUS {
            val += image[x+offset_x, y+offset_y] * basis
        }
    }
//save output for each feature
output[f, x, y] = val
}

//histogram (256 bins per feature)
//each workgroup builds histogram for portion of the image
for f = 0:NUM_AIM_FEATURES {
    //max and min vals across features
    pixel = (input[f, x, y] - min_val)/(max_val-min_val)
    //save rescaled input to a global array
    scaled_output[f, x, y] = pixel
    //update histogram
    atomic_inc(partial_histogram[f, pixel*NUM_HISTOGRAM_BINS])
}

//sum_partial_histograms
 for f = 0:NUM_AIM_FEATURES {
   hist[thread_id] = part_hist[f, thread_id]
     for i = 1:num_work_groups    {
       //sum up all partial histograms
       //in local memory for efficiency
       tmp_hist[thread_id] += part_hist[f, idx + thread_id]
       idx += NUM_HIST_BINS
    }
   //write to the global array
   hist[f, thread_id] = tmp_hist[thread_id]
}

//compute_saliency
//one thread per output pixel
for f = 0:NUM_AIM_FEATURES    {
  idx = round(input[f, x, y]*NUM_HIST_BINS)
  temp -= log(hist[f, idx]/(img_w*img_h) + 0.000001f)
}
output[x, y] = temp
```

Figure 10: Objects in Canabalt: top row shows several key-frames from character animation, bottom row shows various non-lethal objects, robot drill is shown on the right.

- inhibit 20×20 area around the local maximum to avoid selecting too many points close to one another;
- repeat the process until 100 points are found or no values above threshold are left.

Around each local maximum 50 normally distributed points are sampled (coordinates are precomputed on CPU since GPU does not support random number generation). These random points are the centers of 30×30 image patches, which are passed to a convolutional neural network (CNN). The GPU implementation of CNN is based on the DeepLearnToolbox [57]. Only the CNN classifier part of the code actually runs on GPU, while training is done in Matlab offline on CPU. CNN must be able to distinguish patches of 4 classes: runner, non-lethal obstacles (crates), lethal obstacle (drill) and everything else (Figure 10).

Each class has considerable amount of variation. For example, runner's animation is composed of 38 different frames, and sometimes could be confused with shards of glass and flocks of birds. There are also 7 types of non-lethal obstacles - crates, boxes and office furniture. Robotic drills are the least varied in appearance, but they also produce a lot of flying debris, which do not affect the runner but occlude the view of the drill itself and nearby objects. Since template matching and SVMs were not very good at separating these classes, CNN is used instead [43].

An architecture for the network was derived experimentally and has 4 layers - 2 convolution and 2 subsampling (Figure 11). The network was trained for 400 epochs with alpha set to 1.0 and batch size of 50. The training data contained 15000 samples for "other" class and 5000

for each of the runner/crate/drill classes. The final network accuracy was 98%.

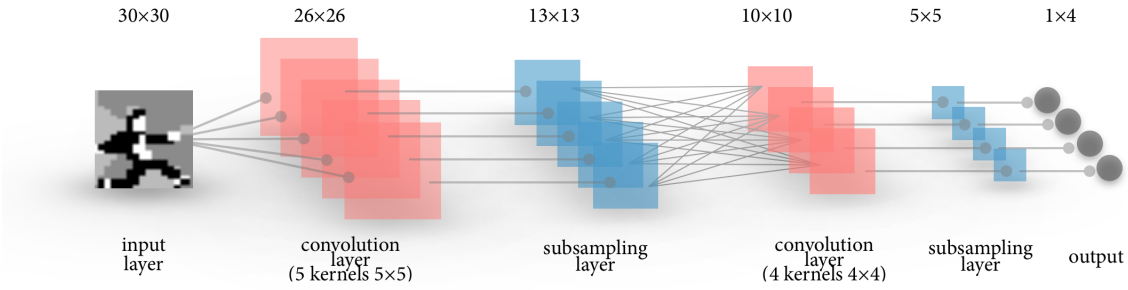


Figure 11: Convolutional Neural Network used for object classification

In general, the best performance on GPU is achieved when the dimensions of arrays are multiples of 32, 64 or 128, which optimizes memory accesses. However, in our case the 16×16 patch is not large enough to discriminate between objects in the game and 32×32 captures too much of the background. As a result, we use patches of size 30×30 pixels, which is optimal for recognition, but not optimal for performance.

Since a full feed-forward pass through CNN can not be done efficiently in a single kernel on GPU, the computation is split into 3 kernels:

`cnnff1` - computes layer 1 by convolution of every 30×30 patch with five 5×5 kernels;

`cnnff2` - computes layer 2 by downsampling patches in layer 1 with no interpolation;

`cnnff3` - computes layer 3 by convolving smaller patches from layer 2 with four 4×4 kernels, also computes layer 4 by downsampling layer 3 result with no interpolation.

For each sample all elements of layer 4 are concatenated in one row to compute feedforward pass into output perceptrons, which gives 4 float values representing probability of the patch belonging to each of the classes. The class for each sample is assigned based on the index of the largest of four probabilities.

Finally, we cluster all points with the same class labels using mean shift [23]. This step is needed because AIM maxima do not necessarily correspond to the centroid of a salient object, besides, objects larger than 30×30 (e.g. drills) may produce several salient points. Clustering

is performed on CPU for each class separately with bandwidth of 20 pixels. Clusters containing more than half of points of "other" class are ignored.

All discovered line segments and centroids of objects are then saved in the Blackboard.

EXTERNAL FUNCTIONS. External functions are called by vTE to eliminate the false detections for objects, find current rooftops and estimate speed.

The starting point is the location of the runner, since its location is the most constrained (movement is vertical with slow drift towards the center of the screen as the speed of the game increases, but at most 200 pixels from the left edge). The longest edge directly below the runner is assumed to be the current platform. If there are any objects on the screen, they are used as additional evidence. If the current platform does not extend beyond the right edge of the frame, we look for lines that begin after the current rooftop ends and select the top one.

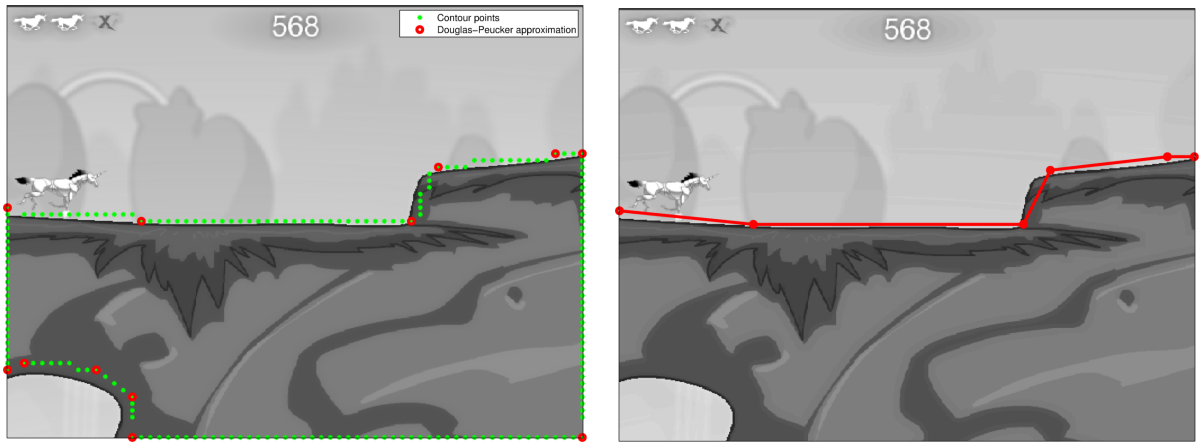
Matching detected objects and finding displacement is done simultaneously. Since many of the obstacles are visually identical, the only way to distinguish between them is by their coordinates. We compute pairwise displacements between all detected objects in the current and previous frames and select globally the most consistent one (or a minimum value if all displacements are unique). The fact that the game scrolls from right to left is an extra constraint used to eliminate incorrect displacements. We also check that the motion of the rooftops is consistent with the object displacements. The speed is estimated using displacements from the past 15 frames.

3.3.4 Robot Unicorn Attack

The only difference in visual processing required for Robot Unicorn Attack is the addition of curve approximation algorithm, since the shape of platforms is more complex than in Canabalt (Figure 12). A sketch of the curve tracing procedure in Cognitive Programs is outlined in [85], however, it was not possible to implement it in real time within our system. In order to find the platform boundaries the following processing steps are taken:

- the original frame is thresholded at 60% of the intensity and resized to 64×64
- connected components are computed using the optimized two-pass method described in [93].

Figure 12: Examples of curved platforms in Robot Unicorn Attack



(a) (b)

Figure 13: a) Screenshot from the Robot Unicorn Attack with contour points (green) detected by the OpenCVBlobsLib, coarse Douglas-Peucker approximation of the surface (red circles), and b) screenshot with detected top of the platform (dotted red line)

- contour points of blobs with area of > 20 pixels are found using the open source library OpenCVBlobsLib[5]
- a coarser polygonal approximation of the blob is obtained by using Douglas-Peucker algorithm [14] (Figure 13a), which depending on the shape and size of the platform, reduces the

[5]Source code available at `http://opencvblobslib.github.io/opencvblobslib/`.

Figure 14: Objects in Robot Unicorn Attack: top row shows several key frames from the unicorn animation, bottom row shows fairies and dolphins (visual distractions, cannot be interacted with), and star is shown on the left.

set of contour points by a factor of 15

- for the remaining contour points the top of the platform is assumed to be the path from the leftmost to the rightmost point (Figure 13b)
- all coordinates are rescaled to the original window size of 512×512.

For the classification step we follow the same steps as in Canabalt. CNN parameters remained the same and the network was retrained on the new set of patches representing objects in the Robot Unicorn Attack (Figure 14). The bandwidth for mean shift is set to 50 to accommodate for larger size of objects.

3.3.5 GPU Performance

Our algorithm largely relies on GPU programming to achieve real-time performance. For example, code written purely in C takes ~ 1 second per frame without the CNN and saliency map construction - two most computationally expensive parts of the algorithm. Our implementation is single-threaded, however, multi-threading on a CPU likely would not achieve a required framerate, hence most time consuming parts of the code were ported to a GPU.

Two AMD FirePro W7000 graphics cards were used to run this algorithm. One was fully dedicated for visual processing and another one was driving the monitor and outputting

Image loading	3.39ms
CPU time	1.2ms
Total GPU time	5.6ms
GPU overhead	1.8ms
Total	11.84

Table 4: Processing times per frame

debugging information in the OpenGL window. Since memory was virtually unlimited for our purposes (W7000 has 4 GB of GDDR5), we only optimized the code to reduce overhead time, limit to a minimum data transfers between CPU and GPU, and increase the performance of each kernel. Although OpenCL has become more popular in the past few years, not many usable libraries are available for visual processing. This dictates the need for implementing all required kernels from scratch and optimizing for the available hardware until satisfactory speed of processing is achieved.

Overall, the algorithm spends ~ 12 msec processing each frame, i.e. the average frame rate is approximately 84 fps (the rate is not fixed and fluctuates between 70 and 90 depending on the frame complexity). About 1/3 of this time is spent on loading a screenshot via XLib. The Hough transform takes about 1 msec on average, most of it spent on sorting indices. Game AI, finding local maxima, mean-shift and other operations take negligible amount of time. The GPU overhead is estimated to be $\sim 15\%$ of the overall processing time.

Often when evaluating the performance of a GPU program, only the actual kernel time is reported, ignoring the fact that running every kernel requires memory allocation, memory transfers, splitting the work among the threads, etc. This time depends on the system (OS, driver version and the video card itself), and may be even longer than the computation itself. In general, a lot of experiments are required to find an optimal way of splitting the work between kernels to minimize the overhead time and fully utilizing available GPU resources.

In our case we reduced the number of kernels from 27 kernels to 21, cutting down the overhead from 25% to 15% of the total processing time. Still, the majority of the kernels perform very simple operations, such as sum reduction or finding min/max in a large array. Although these operations are usually easier to implement on CPU, the overhead from moving

data between CPU and GPU makes it inefficient.

3.4 Game-specific details and helper scripts

3.4.1 Speed estimation

Speed of the game is one of the components required for planning a jump. In both games we measure the speed of the environment, since the character's horizontal coordinate is more or less fixed in the left part of the screen (it drifts very slowly towards the center with time) and only its vertical coordinate varies. In our case the correct speed estimation in online games is complicated by three factors: parallax scrolling, asynchronous sampling and rasterization.

Parallax scrolling creates an illusion of depth in 2D scenes by making the background move slower than the foreground. In Canabalt and many other 2D games parallax scrolling is used to add visual interest to the scene, however, it can also affect accuracy of speed estimation. The problem is in determining the background and eliminating it, so that it does not interfere with the processing.

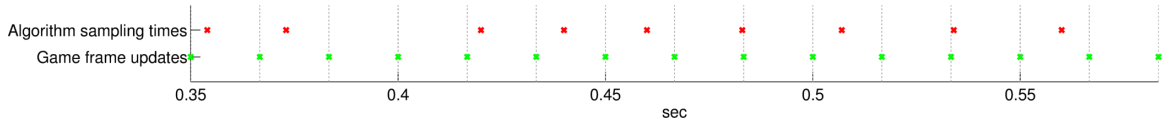


Figure 15: The plot demonstrates asynchronous sampling of the Canabalt game screen by the algorithm. Since Canabalt runs at 60 fps, the screen is updated every $1/60$ sec, this is represented by the green dots. Red dots show the times when the algorithm makes a screenshot of the game.

In Robot Unicorn Attack the background is very distinct and can be easily removed by thresholding. Canabalt is a monochromatic game with little variation in color and complex background with several layers moving at different speeds. In this case simple techniques like thresholding do not apply. Instead, we use landmarks, such as the ends of the rooftops or crates stored in task working memory, to find displacements between the consecutive frames. When the rooftop extends beyond the limits of the screen on both sides and there are no obstacles visible, the speed cannot be determined. In our case it does not cause issues, since

no action would be required on an empty rooftop anyway, therefore we assume that the speed has not changed. When an object or the end of the platform comes into view, the speed is updated as usual.

Another problem, which causes problems for speed estimation is the asynchronous sampling of the game screen. For instance, Canabalt runs at 60 fps and our algorithm takes screenshots at $70 - 90$ fps, but the speed of processing depends on the complexity of the frame and may reduce the framerate to 50 fps for short periods of time (Figure 15). One solution is to fix the processing time for each frame to a certain value. However, in our case it is not optimal. Those parts of vision processing that run on GPU cannot be terminated early, so the time limit would have to be set high to accommodate for the occasional hard frames, which would unnecessarily slow down the algorithm.

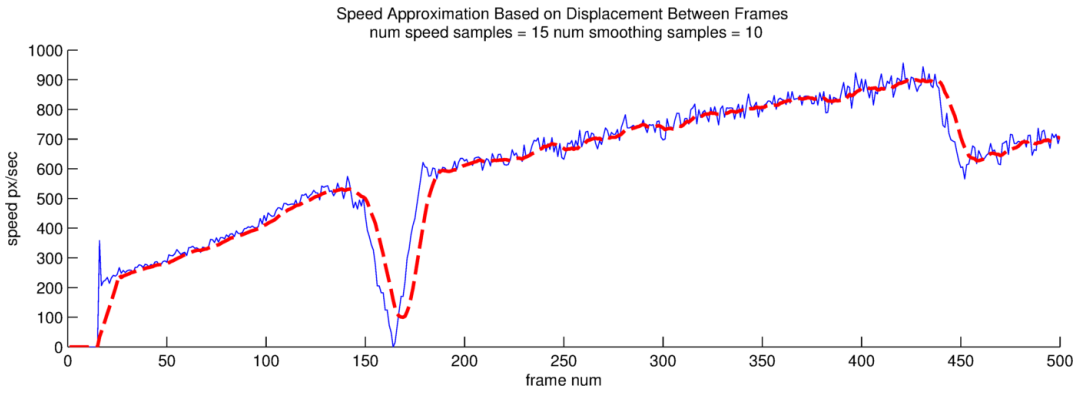


Figure 16: Speed approximation by two-step averaging. The first step is computing the mean instantaneous velocity for each frame based on the 15 previous displacements (blue line). The result of running average (window size 10) applied to the mean instantaneous velocity is shown in red.

A common method of determining speed from the video is by finding instantaneous velocities at n points on the object of interest and averaging them [81]:

$$v = \frac{1}{n} \sum^{n} v_i.$$

The instantaneous velocity is expressed as

$$v_i = \frac{\Delta p}{\Delta t},$$

where Δp is the spatial displacement of the 2D point during the time interval Δt. As a preprocessing step, the velocity samples, that are more than one standard deviation away from the population mean, are removed. Since sampling interval Δt is not constant, applying this method directly to the displacements between frames produces large fluctuations in speed estimation.

On the other hand, the displacement term, Δp, is affected by the rasterization of game graphics, a process of converting vector shapes into a pixel form. Even though the speed of the game increases continuously, rasterization causes displacement values to be rounded to the nearest integer. To reduce the short-term fluctuations we apply a simple moving average over the k speed estimates (in our algorithm we set $k = 10$). Results are shown in Figure 16.

3.4.2 Jump physics

Jumping is the only action available and the final game score largely depends on the ability of the player to make timely and precise jumps. The best strategy is to plan the jump so that the landing spot is as close to the beginning of the next rooftop as possible, because it leaves additional time to react to falling drills and other obstacles. The trajectory of the jump is determined by the initial speed, the amount of time the button was pressed, the location of the runner and the size of the obstacle.

Unfortunately, reverse engineering game physics is not common in the game AI research. Typically, the future state is obtained by running the game engine one or more steps forward. Both games we use are commercial and no source code is available for them, therefore we had to find a way to learn physics from the available data. Conveniently, the physics of Canabalt is not very complex. As we found out from data collected during runtime, when 'X' is pressed, the runner's vertical acceleration is set to a constant value, which does not change for some time

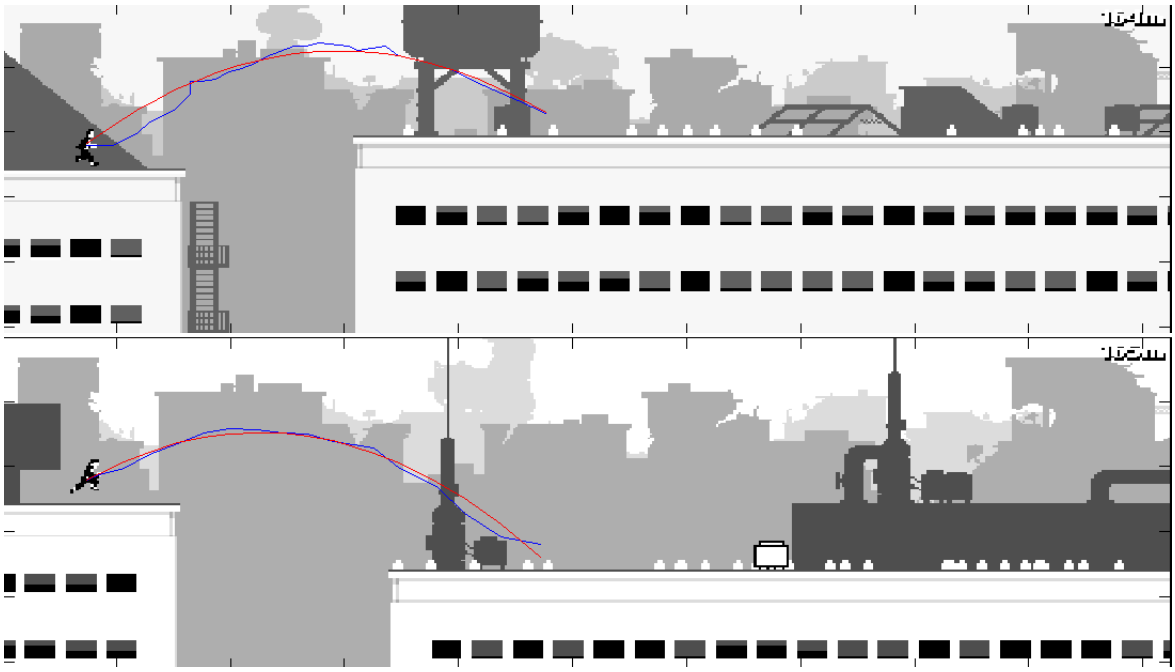


Figure 17: Fitting a parabola to a jump trajectory. Blue line represents collected samples and red line is the fitted curve.

and is then decremented at regular intervals by another constant. We found that a parabola is a good approximation for the trajectory (Figure 17). Holding the button pressed maintains vertical acceleration, and the higher the speed, the higher the jumps become. Pressing the button for more than 350 msec will have no additional effect. Also, consecutive jumps or jumps that immediately follow hitting a crate, are usually lower.

In order to learn how a button press affects the trajectory of the jump we gathered runtime statistics on hundreds of jumps (raw coordinates of the runner in the frame, time stamps for each frame, whether the button was pressed, parameters of the parabola fit to the raw data and speed) and looked for significant correlations between different types of data. We found a 95% correlation between the height of the jump and the duration of the button press and 97% correlation between the speed and maximum height of the jump. The y-component of the jump can be written as

$$y = a \cdot x_{\text{target}}^2 + b \cdot x_{\text{target}}, \tag{1}$$

where a and b are the parabola parameters and t is the time. This function reaches its maximum when

$$x_{\max} = -\frac{b}{2a} \tag{2}$$

The value of $a$can be derived from the current speed from collected jump trajectories:

$$a = A_1 \cdot \text{current\_speed}^2 + A_2 \cdot \text{current\_speed} + A_3 \tag{3}$$

where $A_{1,2,3}$ are parameters found by fitting a polynomial to the distribution. Then we compute

$$b = \frac{(y_{\text{target}} - a \cdot x_{\text{target}}^2)}{x_{\text{target}}} \tag{4}$$

and plug in values of a and b in (2) to find height at $x_{\max}$. Once the height is found, we check if this height can be reached at current speed. If yes, the button press time is returned, if not, it usually means that the jump was planned too early. In this case, if the runner is still far from the obstacle, the function returns 0 and attempts to jump later. Otherwise, if the obstacle is too close, the biggest possible jump is performed.

We apply a similar technique to learn jump parameters for Robot Unicorn Attack. Unlike Canabalt, it is rendered in a square window and since the lookahead is much smaller, most of the jumps are made when the next platform is not visible. To handle such situations the button is pressed when the unicorn is close to the edge of the platform and held until the next platform appears in the field of view. Once it appears, for each successive frame we use learned jump parameters to estimate whether the platform can be reached if the button is released immediately.

3.4.3 Collecting jump trajectories

To collect jump samples for training we save each frame as a png file and record runtime statistics (the coordinates of objects, current speed, displacement, time stamps, whether the button was pressed, whether the character is on the platform, etc.). A MATLAB script is

used to compute displacements between consecutive frames and identify sequences of frames containing a jump. The sequence starts when a jump button is first pressed and ends when the character lands on the next platform. For each jump sequence we recover the true jump trajectory by combining the character's coordinates and displacements and fit a parabola to the vertical component of the trajectory using RANSAC [19]. As a result each jump is characterized by 2 parabola parameters, the duration of the button press and max height. Series with less than 10 frames are discarded, since the noise in the data becomes prevalent and makes fitting a parabola impossible. Regression analysis is also used to find the dependency between max height and button press and between the speed and parameters of the trajectories. After a batch of 50-70 frames is added to the database, we run regression on the data and update jump parameters of the AI code, which is then recompiled.

Because of the random nature of the game and the fact that jumps are infrequent events (on average about 10 jumps are made per 1000m), it was hard to get enough representative samples for a possible range of speed values and button press values.

3.4.4 Game Over detection

GAME OVER screen contains the score and reason for the current session ending, so it is necessary to detect and save the statistics for future analysis. Most games, including the ones we use, do not start a new session automatically after the previous one ended, and require a player to click on the screen or press a button. Having a detection mechanism for the GAME OVER condition allows to run multiple experiments without having to manually start the new session and reset the algorithm parameters. Canabalt also can be paused, at which time the screen is partially covered by a text message and cannot be processed.

Canabalt

Since it is necessary to check for the GAME OVER and PAUSE conditions at each frame, the detection method should be as efficient as possible. In our case the decision is based on the number of white pixels in the frame, since this quantity is computed as part of early visual processing. Frames with more than 30K white pixels are assumed to be GAME OVER, frames

(a)

(b)

(c)

(d)

Figure 18: Screenshots of the paused game (a) and final screens with various reasons for losing(b,c,d)

with more than 12K pixels are PAUSE (Figure 18a) and the rest should be processed fully. After the first GAME OVER frame is detected, all parameters of the algorithm are reset, the screenshot of the last frame is saved and the game is restarted by pressing the 'X' button. When the game is paused the algorithm stays idle.

Saved GAME OVER screenshots are later parsed by a MATLAB script to extract the score and the reason for losing: falling from the rooftop, hitting a wall or crashing into a robotic drill (Figure 18). First, connected components are computed for white pixels in the frame. The largest components are assumed to be the letters of GAME OVER and smaller components directly below are presumed to contain the final score. Finally, each 15×15 patch around the centroids of the small connected components is parsed using CNN.

Robot Unicorn Attack

The game cannot be paused once it has started. As in Canabalt the game is over when the player-controlled character hits a fatal obstacle (bump on the platform or a star) or falls into the gap between platforms. In Robot Unicorn Attack a player has three attempts and the final score is a sum of the scores from three runs. When the game is running, between one and three silhouettes of unicorn are visible to indicate the number of lives remaining (Figure 19). In order to find out whether the game has ended we simply check for white pixels in the top-left corner of the screen.

3.4.5 Debugging

Most of the debugging is done offline using the saved frames, however, saving png files for every frame slows the algorithm down to about 30-40 fps and affects its behavior. In order to observe the algorithms performance in real-time a small OpenGL application runs alongside the browser window and outputs the image of the screenshot with the following information: colored dots at the locations where patches were collected for CNN classification, centroids of objects marked by color-coded crosses, lines for detected rooftops (green for the rooftop under the runner, yellow - next rooftop to land on, blue - all other lines), current speed estimate and frame rate. Dots and crosses representing classification results are color-coded: green for

Figure 19: Screenshots of Robot Unicorn Attack with different number of "lives" left shown in the top-left corner

runner/unicorn, red for fatal obstacles (robotic drills or stars), blue for non-fatal obstacles (crates, boxes and fairies) and yellow for the patches identified as background. A pink cross appears at the top left corner when the control button is pressed.

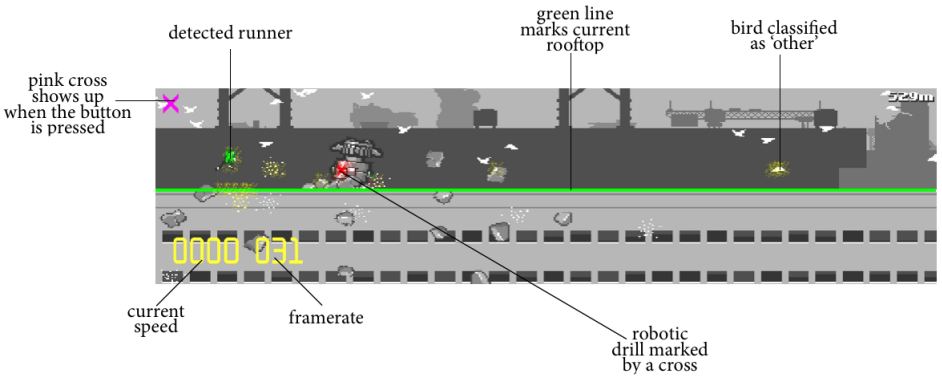


Figure 20: Debugging window showing the algorithm information in real time

Additional GPU-CPU transfers required for updating OpenGL window account for approximately 5% of processing time per frame, but it does not affect the overall performance in terms of the final game score.

4 Evaluation

Playing video games has many subjective aspects, for example, style, strategy and entertainment value, which are hard to quantify, but are important factors when judging performance. To our knowledge, there is no software that can play our games, nor can other existing game playing algorithms be easily adopted to our task. Therefore, we use the final score as our main evaluation metric and measure the performance of the vision module separately.

4.1 Canabalt

Objects and obstacles detection/recognition are crucial for proper functioning of the game logic module. We used 5000 frames from several recorded games with ground truth data generated by hand with locations of runner/crates/robots/rooftops and displacements between neighboring frames.

The runner is correctly detected in 98% frames. The ground truth marks the middle of the character as its location. We consider detection successful if the difference between the runner coordinate determined by the algorithm is within 5 pixel circle centered at the middle of the character (which is about 1/4 of its height). For crates and robot drills the detection rate is also high at $\sim 97\%$, detection is considered successful if it is within the object contour. Most of the mis-detections are caused by sudden shakes of the game screen when a robotic drill falls on the rooftop or a rocket passes in the background (Figure 21).

To measure the baseline performance we modified the algorithm to press the button between 0.05 and 0.35 sec at random intervals (up to 3 seconds) regardless of the current state and recorded 100 games. The average score with this strategy is 151.21m (minimum 101m and maximum 437m).

On average a normal game lasts 48 seconds (approximately $4,000$ frames). Most of the time the runner stays on top of the roof or in the air and does not require any guidance. Only $15 - 20\%$ of frames are critical for successfully playing the game: $10 - 15$ frames before each obstacle to determine the speed correctly and have enough time to press the jump button.

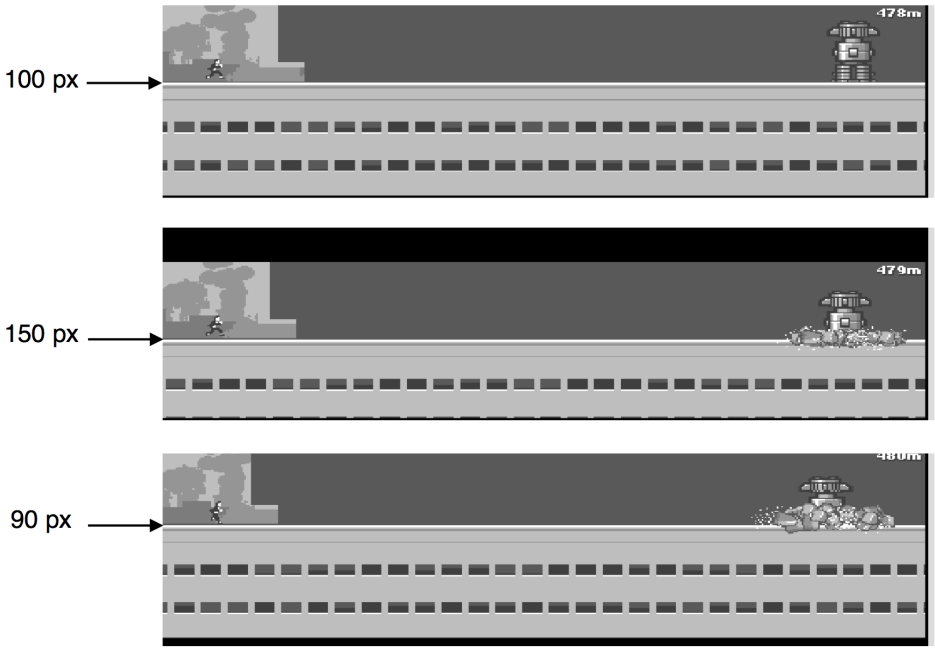


Figure 21: The effect of the fallen robotic drill shown in three consecutive frames. When the drill lands on the platform the image within the game screen starts moving up and down (black is the background of the window). Notice the significant change in the horizontal coordinates of the platform in every screenshot.

However, the algorithm still must be stable enough to identify and track multiple objects for several thousands of frames.

This project would be incomplete without discussing the actual game playing skills of the algorithm. Since it is based on a model of human visual attention, it makes sense to compare it to human players.

Canabalt was released in 2009 and remains popular, with versions for browsers, iOS, Android and BlackBerry available. Each version has slight differences from the original browser game.There are differences in input method (touch screen or keyboard) and the size of the screen (on mobile devices game runs at a higher resolution so there is less vertical panning and

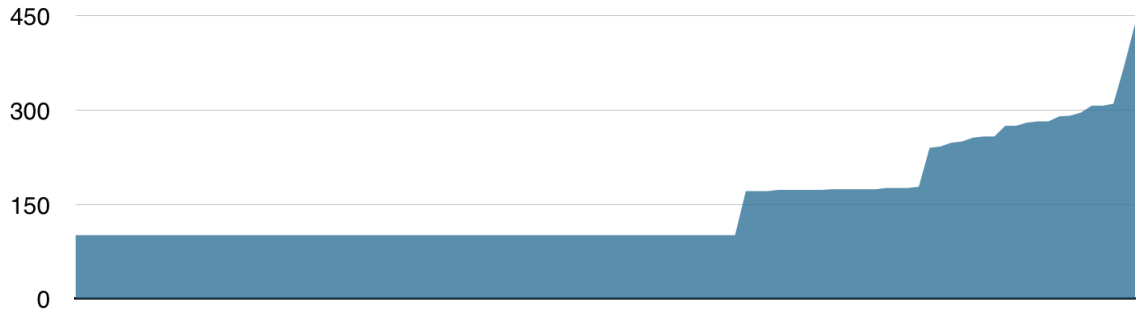


Figure 22: Scores for games played with random action selection strategy

more lookahead). There is no official leaderboard, instead mobile versions of Canabalt allow users to post their scores via Twitter. For example, this unofficial scoreboard contains 57907 scores submitted in 2009-2010 (peak of the game's popularity). The average run was 4037m and the top score was 41785m. Another analysis of the Twitter-submitted scores up to 2011 looks at the best platform to achieve high score and the top reasons for losing (see Figure 23). It shows that the highest score of $>$ 40000m and also the highest average and median scores were reached on iPad. The most common reason for losing was missing a window.

More recent statistics from the browser version of Canabalt are available at `kongregate.com`, which is what we use for evaluation. The all-time highest score reported there is 30700m. Statistics are updated daily and only 100 top scores (one for each user) are displayed. All scores are reset at the end of the week. Because of this the average fluctuates from week to week, but is usually around 2500m. It can be concluded from this statistics that an expert player should be able to score at least 10000m.

We collected statistical data on thousands of games played by our algorithm, including the final score and a reason for failing (in browser version it can be either one of three: hitting a wall, falling into a gap or hitting a robot drill). Mean score of the last 1000 games was $>$ 3000 and top score was 25,254m, making it #18 in the all-time best ranking posted on `kongregate.com`. The most common reason for failing was hitting a wall due to the mistakes while jumping. As mentioned before, jumping is the essential part and our algorithm has much better control over the keyboard than a human player. It is tuned to select the smallest

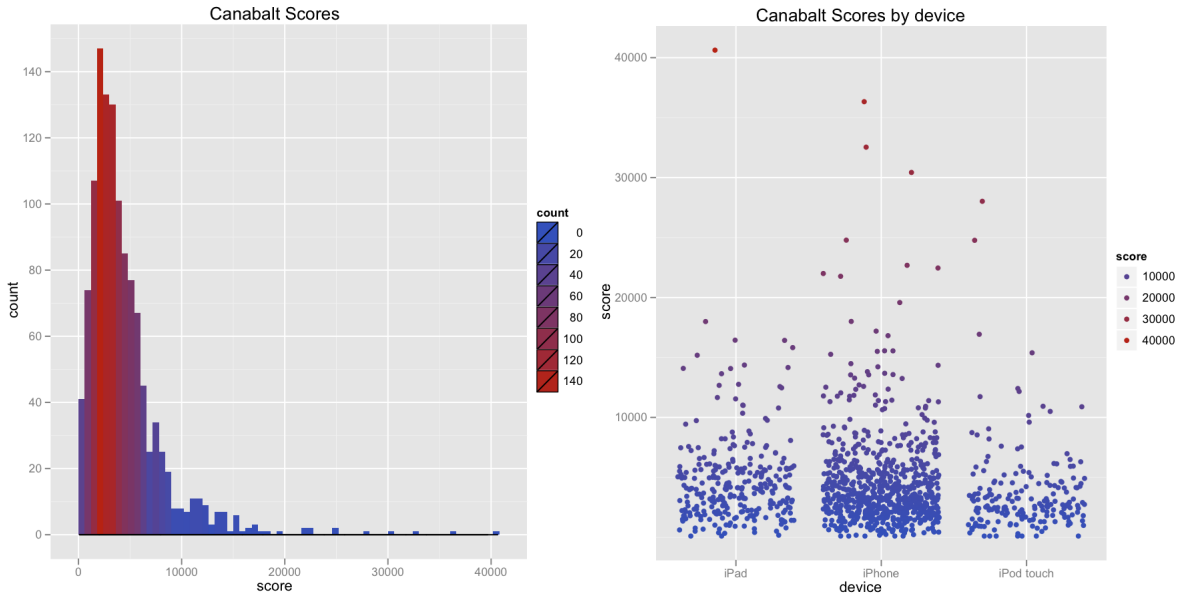


Figure 23: Distribution of scores for Canabalt games played online

parabola to land close to the edge of the rooftop and even several milliseconds of delay can have a significant effect. Figure 24 shows the distributions of button press times for the human player (myself) and the algorithm: on a physical keyboard the average press time is 180 msec, while the mean for the algorithm is 92 msec.

4.2 Robot Unicorn Attack

Robot Unicorn Attack is visually simpler than Canabalt: the camera movement is smooth, there are only 3 types of objects distinct from the background and fewer distractors. The detection accuracy for the unicorn is at 95% based on 5000 frames collected from several games (if the cluster center is within 7 pixel radius (1/4 of its height) from the center of the unicorn's torso). The mis-detections mostly happen when the unicorn is dashing and temporarily disappears behind the explosion, however, it does not affect the performance since the controls are inactive at this time.

The final score in Robot Unicorn Attack depends on the time spent playing and also on

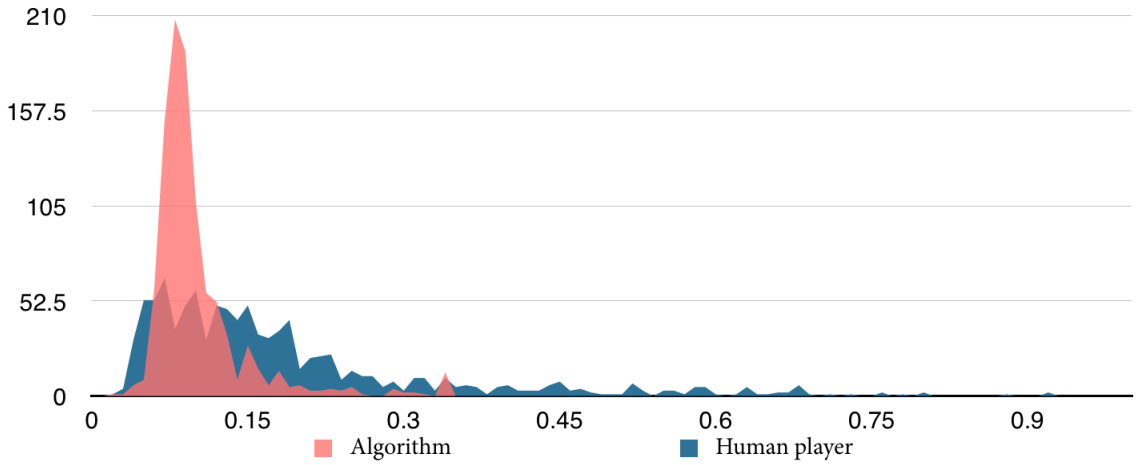


Figure 24: The histogram of button press times (100 games each)

Figure 25: Screenshots showing changes in appearance of the unicorn when dashing through the star

the bonus points from collecting fairies and dashing through the stars. The first star is worth 100 points and every next star increments the amount by 100 points. If one star is missed, the next bonus will be again 100. Points for fairies are assigned similarly, except the counter starts at 10 and is incremented by 10 every time a next fairy is collected. The points for simply staying alive amount to approximately 1000 points per 10 seconds of gameplay. Obviously, collecting all stars and fairies without breaking the sequence while staying alive as long as possible would bring the most points. However, the stars may be placed on the island above or below the unicorn and may not be accessible.

The easiest and safest strategy is based on following the fairies, which indicate the safest trajectory for jumping between the platforms. When the star is on the same level as the

Figure 26: An example of a platform with several layers in Robot Unicorn Attack

unicorn, dash through it, otherwise it is safer to ignore it.

This game had over $32,000,000$ plays on the popular gaming site adultswim.com since its release in 2010. Unfortunately, the official scoreboard has recently been removed by the maintainers of the game. Due to the game's popularity there exist many ways of tampering with the game allowing any user to submit an arbitrary final score, meaning that the data found on other score aggregators is generally not reliable (e.g. scores of several billion's of points). However, based on the interview with the game's creator[6], videos of expert players posted online and our own experience it is safe to assume that top players are able to achieve around $100,000$ points per run and close to $300,000$ for a game. Expert players get approximately $80,000 - 100,000$ points for a game.

We did not spend as much time tuning the algorithm to improve its score as we did for Canabalt and stopped once it was able to reach more than $30,000$ points in a single run. We believe that the results can be further improved by fully utilizing the control mechanisms in the game, for example, a double jump (i.e. jump again in the air to correct the trajectory).

By adding a second game of similar genre we were able to test the flexibility of the Cognitive Programs. Few changes were required in order to make the system play a new game. Namely, the addition of visual processing routines to handle curved platforms, retraining CNN for different types of objects and changes in the control function to add an extra key for dashing.

[6] http://gfbrobot.com/2011/06/22/designing-robot-unicorn-attack-an-interview-with-scott-stoddard/

5 Discussion and Suggestions for Future Work

The main goal for this thesis was to demonstrate that the Cognitive Programs framework would be sufficient for complex visual tasks. Since building a whole system was not feasible given the time and resources available, we decided to implement only a subset of CPs needed to perform a single non-trivial task. The task was playing an online video game with minimal controls, where visual processing plays a crucial role. The game we trained our algorithm to play is called Canabalt. It is an endless scroller, where environment is randomly generated for every session. Later we extended our algorithm to play another game of the same genre - Robot Unicorn Attack.

We came across multiple theoretical and engineering issues in the process of implementing Cognitive Programs and while studying extensive literature on cognitive architectures and past projects related to visual routines.

One of the first problems we had to address was extending the Cognitive Programs to work in dynamic environments. The original concept of CPs (Figure 3a) was designed primarily for static stimuli. We had to find a way of representing motion flow and matching recognized objects between the frames within the existing framework. In our implementation all elements critical for gameplay (object locations, speed, etc.) are placed into vWM and tWM and later retrieved by external methods to compute displacement between the frames or match objects between frames. It demonstrates the functional importance of working memory, which is prominent in the human visual system and was also reflected in the past implementations of visual routines. However, a more biologically plausible solution for this problem is needed.

Timing of various processes is another important problem not outlined in the initial concept. It is assumed that some of the components of CPs run in parallel, mimicking the human visual system, but the diagram only shows the connections between elements and direction of the information flow. We had engineering issues while trying to use GPU programming in a multi-threaded context, therefore our program runs on a single thread. This avoids the issue for now, but any future implementation of Cognitive Programs would have to address

the synchronization of various concurrent processes.

The problem of finding data structures for describing the task and passing information within the framework also remains open-ended. To the best of our knowledge this problem is still largely unresolved in the literature and hardcoding the solution is the only viable route at present. Finding a suitable representation is tied to another open research area of learning how to combine and tune these representations for various tasks. Past research demonstrates that learning a sequence of visual routines is a hard problem even in the simplest cases with a handful of parameters. For Cognitive Programs this task becomes much more complicated, because every method can be parameterized during runtime depending both on the task and changes in the environment. For example, for our algorithm we had to tune more than 20 parameters by hand, including various thresholds for visual processing, parameters of CNN and AIM, gameplay and game physics variables.

Apart from conceptual problems we had an additional requirement of realtime performance. The task of playing an online game puts an upper bound on a computation of 20 ms per frame. The only way to achieve this is to move as much processing as possible to a GPU, which in turn introduces a number of restrictions. For instance, some parameters of GPU kernels cannot be changed during runtime. In our case computing saliency using AIM and recognition with CNN both rely on local memory, which has to be allocated at compile time. So the sizes and the number of filters for AIM and CNN cannot be changed on the fly without recompiling the kernel. Besides, there are limitations on the amount of memory available and memory accesses (e.g. images cannot be changed in place), recursion is not allowed, for efficiency some of the serial operations must be done in parallel (e.g. overt changes of attention to analyze various objects in the fovea), etc. These requirements forced deviations from the original concept of Cognitive Programs. With better hardware and drivers it would be conceivable to implement it fully. On the other hand, these are the issues that must be solved by any real-time biological implementation.

Most of the past research on visual routines worked in simulated or controlled environments, only a few projects attempted explicit visual processing. We originally planned to use a camera

to take an image of the computer screen instead of programmatically capturing screenshots. Figure 27 shows an example of a screenshot taken with a camera. The issues with using the camera input were motion blur and artifacts introduced by pixels on the screen, uneven illumination and distortions from camera lens. Motion blur becomes less prominent when the frame rate is high enough (our camera could output 60 fps at 1024×256), possibly with a better and faster camera it can be somewhat resolved. We plan to return to camera images and think of what modifications to our algorithm would be needed to compensate for additional noise and artifacts.

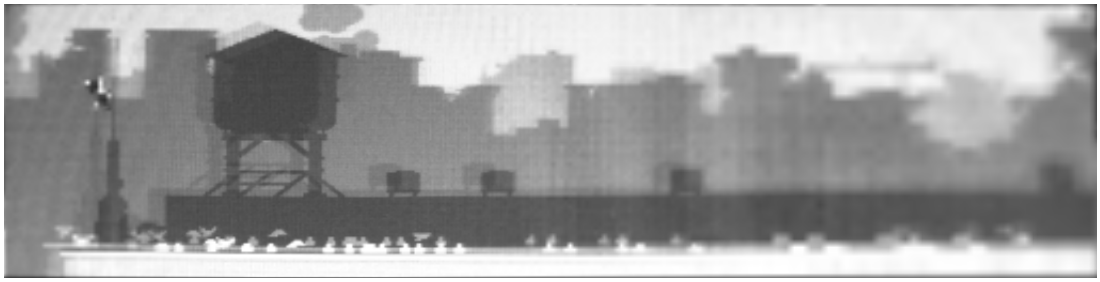


Figure 27: An example of a typical image from a camera with added foveation

Although we have a working system capable of playing two games on an expert level, its performance could be further improved. For example, implementing optical flow with larger number of points and adding segmentation of the scene would greatly increase accuracy and stability of the algorithm. In particular, it could solve one of the issues, that occasionally causes the algorithm to fail: when falling shards of glass or exhaust from the rocket in the background are mistakenly identified as part of the rooftop, the displacement between current and previous frame is wrong and in turn affects the estimate for the trajectory of the following jump.

Overall, we believe that our experience of building a first working prototype of Cognitive Programs is an important step toward a full implementation. Even though our visual executive is specific to playing a particular type of games, the low-level visual operations - edge detection, clustering, object localization and identification - are general enough to be applied to a broader range of tasks.

An algorithm that can visually analyze and play a video game may also benefit the gaming AI research. Firstly, there is a lot of interest in developing AI to imitate human performance. Having a system based on human vision and providing it with the same information available to a human player may be helpful in achieving this goal. Secondly, much of the game AI research relies on emulators, clones or open-source games to provide noise-free data for learning algorithms, meaning that the majority of modern commercial games are not available for experimentation.

Finally, a framework for performing complex visual tasks has a direct application in mobile robotics, especially for tasks involving interaction with the environment in real time.

References

[1] Philip E Agre and David Chapman. Pengi: An Implementation of a Theory of Activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, volume 278, pages 268–272, 1987.

[2] J Albus, C Schlenoff, R Madhavan, S Balakirsky, and T Barbera. Integrating Disparate Knowledge Representations Within 4D / RCS. In *Proceedings of the 2004 AAAI Fall Symposium*, 2004.

[3] James Sacra Albus. RCS: A reference model architecture for intelligent systems. Technical report, 1995.

[4] John R Anderson, Daniel Bothell, Michael D Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An integrated theory of the mind. *Psychological review*, 111(4):1036–1060, 2004.

[5] Raul Arrabales, Agapito Ledezma, and Araceli Sanchis. CERA-CRANIUM : A Test Bed for Machine Consciousness Research. In *International Workshop on Machine Consciousness*, 2009.

[6] D. Paul Benjamin and Damian Lyons. Toward A Cognitive Computer Vision System.

[7] Slawomir Bojarski and Clare Bates Congdon. REALM: A rule-based evolutionary computation agent that learns to play Mario. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games (CIG)*, 2010.

[8] David H. Brainard. The psychophysics toolbox. *Spatial vision*, 10, 1997.

[9] Rodney Brooks. A Robust Layered Control System for a Mobile Robot. Technical report, MIT, 1985.

[10] Neil D.B. Bruce and John K. Tsotsos. Saliency Based on Information Maximization. In *Advances in Neural Information Processing Systems 18 (NIPS 2005)*, 2005.

[11] Nicholas Cassimatis. A Cognitive Substrate for Achieving Human-Level Intelligence. *AI Magazine*, 27(2):45–56, 2006.

[12] Nicholas L. Cassimatis, J. Gregory Trafton, Magdalena D. Bugajska, and Alan C. Schultz. Integrating cognition, perception and action through mental simulation in robots. *Robotics and Autonomous Systems*, 49(1-2 SPEC. ISS.):13–23, 2004.

[13] David Chapman. Vision, Instruction and Action. Technical report, MIT, 1990.

[14] David H. Douglas and Thomas K. Peucker. Algorithms for the Reduction of the Number of Points Required To Represent a Digitized Line or Its Caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

[15] Mark Drummond, John L. Bresina, and Smadar T. Kedar. The entropy reduction engine: Integrating planning, scheduling, and control. Technical report, 1991.

[16] Rick Evertsz, Matteo Pedrotti, Paolo Busetta, Hasan Acar, and Frank E. Ritter. Populating VBS2 with Realistic Virtual Actors. In *Proceedings of the 18th Conference on Behavior Representation in Modeling and Simulation*, number April, pages 1–8, 2009.

[17] Jeffrey Fayman, Ehud Rivlin, and Henrik I Christensen. A System for Active Vision Driven Robotics. In *Proceedings of IEEE International Conference on Robotics and Automation*, 1996.

[18] Jeffrey A. Fayman, Ehud Rivlin, and Henrik I. Christensen. AV-Shell, an Environment for Autonomous Robotic Applications Using Active Vision. *Autonomous Robots*, 6(1):21–38, 1999.

[19] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[20] K.D. Forbus, M. Klenk, and T. Hinrichs. Companion Cognitive Systems: Design Goals and Lessons Learned So Far. *IEEE Intelligent Systems*, 24(4), 2009.

[21] Stan Franklin. A Foundational Architecture for Artificial General Intelligence. *Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms*, pages 36–54, 2007.

[22] Stan Franklin and F.G. Patterson. The LIDA architecture: Adding new modes of learning to an intelligent, autonomous, software agent. In *Integrated Design and Process Technology*, pages 1–8, 2006.

[23] K. Fukunaga and L. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory*, 21(1):32–40, 1975.

[24] Salgian Garbis. *Tactical Driving Using Visual Routines*. PhD thesis, 1998.

[25] Ben Goertzel, Hugo De Garis, Cassio Pennachin, and Nil Geisweiller. OpenCog Prime: a Cognitive Synergy Based Architecture for Embodied Artificial General Intelligence. *Proceedings of ICCI-09*, pages 1–12, 2009.

[26] Neil S Halelamien and Advisor David S Touretzky. *Visual Routines for Spatial Cognition on a Mobile Robot*. PhD thesis, 2004.

[27] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A Neuroevolution Approach to General Atari Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, PP(99):1–1, 2014.

[28] Jeff Hawkins and Dileep George. Hierarchical temporal memory: Concepts, theory and terminology. Technical report, 2006.

[29] B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot, and M. Balabanovic. A domain-specific software architecture for adaptive intelligent systems. *IEEE Transactions on Software Engineering*, 21(4):1–43, 1995.

[30] Mary Hayhoe. Vision Using Routines: A Functional Account of Vision. *Visual Cognition*, 7(1-3):43–64, 2000.

[31] Ian Horswill. Visual Routines and Visual Search: a Real-Time Implementation and an Automata-Theoretic Analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[32] Michael Patrick Johnson and Stephen a Benton. *Evolving Visual Routines*. PhD thesis, 1993.

[33] Michael Patrick Johnson, Pattie Maes, and Trevor Darrell. Evolving Visual Routines. In *Proceedings of Artificial Life IV Conference*, 1994.

[34] Marcel Adam Just and Sashank Varma. The organization of thinking: what functional brain imaging reveals about the neuroarchitecture of complex cognition. *Cognitive, affective & behavioral neuroscience*, 7(3):153–191, 2007.

[35] S Kalyanakrishnan and D Kalyanakrishnan. IeorIITB2048. Technical report, Indian Institute of Technology, 2015.

[36] Sergey Karakovskiy and Julian Togelius. The Mario Ai benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):55–67, 2012.

[37] David E. Kieras and David E. Meyer. An Overview of the EPIC Architecture for Cognition and Performance With Application to Human-Computer Interaction. In *Human-Computer Interaction*, pages 391–438. Lawrence Erlbaum Associates, 1997.

[38] John R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, 1994.

[39] Wouter Kruijne and John K Tsotsos. Visuo-cognitive routines as a framework for visual cognition Visuo-cognitive Routines. Technical report, 2011.

[40] Peter C. R. Lane, Anthony Sykes, and Fernand Gobet. Combining Low-Level Perception with Expectations in CHREST. In *Proceedings of the European Cognitive Science Conference (EuroCogSci)*, pages 205–210, 2003.

[41] Pat Langley, Dongkyu Choi, and Nishant Trivedi. ICARUS Users Manual. Technical report, 2011.

[42] Pat Langley, Kathleen B. McKusick, John A. Allen, Wayne F. Iba, and Kevin Thompson. A design for the ICARUS architecture. In *Proceedings of the AAAI Spring Symposium on Integrated Intelligent Architectures*, 1991.

[43] Yann LeCun and Yoshua Bengio. Convolutional Networks for Images, Speech, and Time-Series. *The Handbook of Brain Theory and Neural Networks*, 1995.

[44] Damian M. Lyons and Michael A. Arbib. Formal model of computation for sensory-based robotics. *IEEE Transactions on Robotics and Automation*, 5(3):280–293, 1989.

[45] Benjamin D. Paul Lyons D.M. A relaxed fusion of information from real and synthetic images to predict complex behavior. *Proceedings of The International Society for Optical Engineering (SPIE)*, 8064, 2011.

[46] D. Marr and T. Poggio. A computational theory of human stereo vision. In *Proceedings of the Royal Society of London. Series B, Biological Sciences*, volume 204, pages 301–328, 1979.

[47] Michael Mateas. Expressive AI : Games and Artificial Intelligence. In *Proceedings of the 2003 DiGRA International Conference*, 2003.

[48] Andrew Kachites McCallum. Learning to Use Selective Attention and Short-Term Memory in Sequential Tasks. In *From Animals to Animats, Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, 1996.

[49] Andrew Kachites Mccallum. Learning Visual Routines with Reinforcement Learning. Technical report, 1996.

[50] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei a Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[51] J. William Murdock and Ashok K. Goel. Localizing Planning with Functional Process Models. In *Proceedings of the 13th International Conference on Automated Planning & Scheduling (ICAPS 2003)*, 2003.

[52] William Murdock and Ashok Goel. Meta-case-Based Reasoning : Using Functional Models to Adapt Case-Based Agents. In *Proceedings of the 4th. International Conference on Case-Based Reasoning (ICCBR'01)*, pages 407–421, 2001.

[53] Enkhbold Nyamsuren and Niels A Taatgen. Pre-attentive and Attentive Vision Module. *Cognitive Systems Research*, 24:62–71, 2013.

[54] Gary Ogasawara. *RALPH-MEA: A Real-Time, Decision-Theoretic Agent Architecture.* PhD thesis, UC Berkeley, 1993.

[55] Santiago Ontanon, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(4):293–311, 2013.

[56] Randall C. OReilly, Dean Wyatte, Seth Herd, Brian Mingus, and David J. Jilk. Recurrent processing during object recognition. *Frontiers in Psychology*, 4, 2013.

[57] Rasmus Berg Palm. *Prediction as a candidate for learning deep hierarchical models of data.* PhD thesis, 2012.

[58] B. Hart P.E., Nilsson N.J., and Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[59] Tom Pepels, Mark H M Winands, and Marc Lanctot. Real-time monte carlo tree search in Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):245–257, 2014.

[60] Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M Lucas, Adrien Cou, Jerry Lee, Chong-u Lim, and Tommy Thompson. The 2014 General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games*, pages 1–15, 2014.

[61] Yusuf Pisan. A visual routines based model of graph understanding. In *Proceedings of the 17th Annual Conference of the Cognitive Science Society*, 1995.

[62] Mihai Polceanu. MirrorBot: Using human-inspired mirroring behavior to pass a turing test. In *Proceedings of IEEE Conference on Computatonal Intelligence and Games, CIG*, 2013.

[63] Rui Prada and Phil Lopes. The Geometry Friends Game AI Competition. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2015.

[64] Rajesh P.N. Rao and Dana H. Ballard. An active vision architecture based on iconic representations. *Artificial Intelligence*, 78(1-2):461–505, 1995.

[65] Satyajit Rao. *Visual Routines and Attention*. PhD thesis, 1998.

[66] Jochen Renz and Xiaoyu Ge. The Angry Birds AI Competition. *AI Magazine*, 36(2):85–87, 2015.

[67] Pieter R. Roelfsema. Elemental operations in vision. *Trends in Cognitive Sciences*, 9(5):226–233, 2005.

[68] Brandon Rohrer. BECCA: Reintegrating AI for natural world interaction. Technical report, 2012.

[69] Brandon Rohrer. An implemented architecture for feature creation and general reinforcement learning. In *Workshop on Self-Programming in AGI Systems. 4th International Conference on Artificial General Intelligence*, 2014.

[70] Marc Romanycia. Visual Routines: Ingredients, Design, and Control. In *Procedings of the Canadian Conference on Vision Interface (VI)*, 1988.

[71] Marc Romanycia. *The Design and Control of Visual Routines for the Computation of Simple Geometric Properties and Relations*. PhD thesis, 1994.

[72] Albert L. Rothenstein and John K. Tsotsos. Attentional Modulation and Selection - An Integrated Approach. *PLoS ONE*, 9(6):1–11, 2014.

[73] Stuart J. Russell. An architecture for bounded rationality. *ACM SIGART Bulletin*, 2(4):146–150, 1991.

[74] Alexei V. Samsonovich, Giorgio A. Ascoli, Kenneth A. De Jong, and Mark A. Coletti. Integrated Hybrid Cognitive Architecture for a Virtual Roboscout. In M Beetz, K. Rajan, M Thielscher, and R.B. Rusu, editors, *Cognitive Robotics: Papers from the AAAI Workshop, AAAI Technical Reports*, pages 129–134. AAAI Press, 2006.

[75] Stéphane Sanchez and Sylvain Cussat-Blanc. Gene regulated car driving: using a gene regulatory network to drive a virtual car. *Genetic Programming and Evolvable Machines*, 15(4):477–511, 2014.

[76] Stuart C. Shapiro and Jonathan P. Bona. The Glair Cognitive Architecture. In Alexei V. Samsonovich, editor, *Biologically Inspired Cognitive Architectures-II: Papers from the AAAI Fall Symposium*. AAAI Press, 2010.

[77] Hans Strasburger, Ingo Rentschler, and Martin Jüttner. Peripheral vision and pattern recognition: a review. *Journal of vision*, 11(5):13, 2011.

[78] Ron Sun. The importance of cognitive architectures: an analysis based on CLARION. *Journal of Experimental & Theoretical Artificial Intelligence*, 19(2):159–193, 2007.

[79] Gheorghe Tecuci and Mihai Boicu. 9 Military Applications of the Disciple Learning Agent. In L. Jain, editor, *Advanced Information Systems in Defense and Related applications.*, pages 1–28. Springer Verlag, 2002.

[80] Gheorghe Tecuci, Mihai Boicu, Cristina Boicu, Dorin Marcu, Bogdan Stanescu, and Marcel Barbulescu. The Disciple-RKF Learning and Reasoning Agent. *Computational Intelligence*, 21(4):1–38, 2005.

[81] M. S. Temiz, S. Kulur, and S. Dogan. Real Time Speed Estimation From Monocular Video. In *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences (ISPRS)*, volume XXXIX-B3, 2012.

[82] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N Yannakakis. The Mario AI Championship 2009-2012. *AI Magazine*, 34(3):89–92, 2012.

[83] David S. Touretzky, Neil S. Halelamien, Ethan J. Tira-Thompson, Jordan J. Wales, and Kei Usui. Dual-coding representations for robot vision programming in Tekkotsu. *Autonomous Robots*, 22(4):425–435, 2007.

[84] J K Tsotsos, S M Culhane, W Y K Wai, Y H Lai, N Davis, and F Nuflo. Modeling Visual-Attention Via Selective Tuning. *Artificial Intelligence*, 78(1-2):507–545, 1995.

[85] John K. Tsotsos and Wouter Kruijne. Cognitive programs: software for attention's executive. *Frontiers in Psychology*, 5(November):1–16, 2014.

[86] S Ullman. Visual routines. *Cognition*, 18(1-3):97–159, 1984.

[87] G.J. J van den Braak, Cedric Nugteren, Bart Mesman, and Henk Corporaal. Fast Hough Transform on GPUs : Exploration of Algorithm Trade-offs. In *Proceedings of the 13th International Conference on Advanced Concepts for Intelligent Vision Systems*, volume 100, page 1, 2011.

[88] Kurt VanLehn and William Ball. Goal Reconstruction: How Teton Blends Situated Action and Planned Action. Technical report, Department of Computer Science and Psychology, Carnegie Mellon University, 1989.

[89] Manuela Veloso, Jaime Carbonell, Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: the PRODIGY architecture. *Journal of Experimental & Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[90] Steven Vere, Hanover Street, and Palo Alto. A basic agent. *Computational Intelligence*, 6:41–60, 1990.

[91] Samuel Wintermute. An Overview of Spatial Processing in Soar / SVS Investigator. Technical Report CCA-TR-2009-01, University of Michigan, 2009.

[92] Calden Wloka. *Integrating Overt and Covert Attention Using Peripheral and Central Processing Streams*. PhD thesis, 2012.

[93] Kesheng Wu, Ekow Otoo, and Kenji Suzuki. Optimizing two-pass connected-component labeling algorithms. *Pattern Analysis and Applications*, 12(2):117–135, 2009.

[94] Weilie Yi and Dana H. Ballard. Routine based models of anticipation in natural behaviors. In *AAAI Fall Symposium, From Reactive to Anticipatory Cognitive Embodied Systems*, 1995.