A flexible, extensible object oriented real-time near photorealistic visualisation system: the system framework design

Anthony Jones and Dan Cornford

Knowledge Engineering Group, School of Engineering and Applied Science, Aston University, Birmingham, B4 7ET, UK d.cornford@aston.ac.uk

Summary. In this paper we describe a novel, extensible visualisation system currently under development at Aston University. We introduce modern programming methods, such as the use of data driven programming, design patterns and the careful definition of interfaces to allow easy extension using plug-ins, to 3D landscape visualisation software. We combine this with modern developments in computer graphics, such as vertex and fragment shaders, to create an extremely flexible, extensible real-time near photorealistic visualisation system. In this paper we show the design of the system and the main sub-components. We stress the role of modern programming practices and illustrate the benefits these bring to 3D visualisation.

1 Background

The visual output of most current GIS visualisation software often exhibits either a low level of visual realism with high levels of user interaction[1], or has a high degree of visual realism with low levels of user interaction[2].

Commercial applications producing computer-based renderings based on geographical information, for example, WorldPerfect[3] and LandXplorer[4] are relatively easy to use, but very difficult to customise to specific requirements, or to add desired features and behaviours. Recently alternatives[1, 2] have been proposed based on modern computer game engines that place a heavy emphasis on geospatial representation such as Microsoft Flight Sim-

[1] For example, the viewpoint may be altered in position and focus in an interactive way, images are often animated (albeit with low quality textures), there is a low number of polygons in the scene, and a small number of objects are depicted in total

[2] For example, the viewpoint position, focus and path are predetermined; the scene contains high fidelity textures, accurate lighting and shadows, and detailed objects

[3] http://www.metavr.com/products/worldperfect/worldperfect.html

[4] http://www.landex.de/

ulator2004[5]). Game engines appear attractive because of the high levels of immersion they are required to produce, and the increasingly realistic quality of the graphics this entails. The development of game engines occurs within a problem domain whose focus greatly influences the constraints and capabilities of the software. For example, a game engine can be written to optimise enclosed environments with a relatively small number of dynamic objects, focussing on the use of immersive lighting and special effects in order to increase user emersion. While it has been demonstrated that existing game engines can be applied to visualisation in a wide range of applications[3], the applicability of game engines will ultimately be restricted by the context of their original problem domains.

The application framework we describe in this paper is based upon a novel design that extends and improves upon an existing model intended for computer games developers[4]. In contrast to the fixed information processing pipeline common to current GIS visualisation software[5], the geospatial rendering system incorporates an information processing pipeline that is highly flexible and extensible. This will be achieved through the combination of plugins and data-driven content and behaviour which will allow the application framework to support a diverse range of applications. Introducing data-driven design to the field of GIS visualisation provides an exciting opportunity to create a uniquely flexible visualisation system. We go on to describe the application framework's use of modern methodologies in order to produce near photo-realistic renderings at interactive frame rates. While a small number of GIS visualisation software developers are currently using modern rendering technology to produce non-realtime images[6], the application of emerging techniques to produce highly detailed, interactive near photo-realistic renderings of spatiotemporal scenes has yet to be realised in the field of GIS visualisation.

A Running Application Example

In order to provide a more concrete illustration of the application framework's capabilities, we present an example problem domain that will be used and extended throughout the paper. The example focusses on the development of a traffic simulation system with the following functional requirements:

- The system will import both Integrated Transport Network (ITN)[7] and topography data in order to inform the modelling of a traffic simulation.
- The system will maintain a real-time traffic simulation occupying the given ITN. The simulation will include dynamic traffic elements (such as traffic lights), a range of vehicle types, and pedestrians. While passive items such as roads and buildings can simply be represented, active items such as

[5] `http://www.microsoft.com/games/flightsimulator/`

[6] For example, see `http://www.3dnature.com/index.html`

[7] Further information: `http://www.ordnancesurvey.co.uk/oswebsite/products/osmastermap/itn/`

traffic lights, vehicles and pedestrians must exhibit appropriate runtime behaviour.
- The system is also required to produce a real-time visual output that will provide the user with an illustrative summary of the simulation's changing state over time.

Additional Applications

It is important to stress here that these example applications are supposed to illustrate the potential of the system, and have not been implemented at this early stage. Our focus is visualisation, but our longer term goal is an integrated modelling environment that is fast, efficient, extensible and flexible. The framework supports user interaction, and so users will be able to orient themselves in a depicted scene by manipulating the camera's position and orientation; similarly, queries and modifications of the modelled environment could be made via an extensible range of input devices. The types of applications we envisage include:

- *Visual assessments* The framework has been designed to support the simulation and visualisation of a wide variety of dynamic spatiotemporal environments, and this fundamental functionality could easily form the basis of a visual assessment application. Due to the use of data-driven programming (as described in Section 2.3), designated study areas can be described incrementally through an XML based (and likely tool-oriented) definition of object properties, types and instances. Visualisations can contain animation, such as wind farm blades that turn, trees that appear to sway in the wind, and crops that grow over time. Visualisations can also include Artificial Intelligence (AI) so that cyclists make use of a proposed bicycle route and pedestrians explore a new shopping center.
  Due to the text-based nature of the framework's scene descriptions, users will be able to quickly influence the detail, realism and overall visual appearance of the study area. Changes can be made to emphasise specific details and modifications (such as a user's home), or to highlight objects with similar properties (all proposed elements, for example).
- *Soil erosion* The framework is designed to be flexible, and thus can also incorporate non-trivial spatiotemporal models, for example it would be possible to produce a simplified erosion model to act on a ground model or Triangulated Irregular Network to model (in a naive manner it must be admitted) erosion and render the output realistically, in accelerated time. The application maintains both a real time clock and a model time clock, so this is very easy to undertake.
- *Process-based models, polling input via the task, input, and binding system* If a more detailed model were required the framework could be readily coupled to a more complex process based model, via the task, input, and

binding systems (see Section 2.2). A good example of this might be the coupling of the simulation world with a numerical weather prediction model to provide realistic weather conditions with the correct timing and location with respect to the model forecast.

2 The Application Framework

We are developing an application framework that will form the basis of a modelling and visualisation environment, where the client is able to tailor the application according to their own requirements through the use of customisations of, and extensions to the framework's runtime behaviour. Figure 1 illustrates the core components that make up the application framework. The framework's overall design is loosely based on the design of an object composition framework presented in [4]. The framework separates overall application functionality into a number of coherent, loosely coupled responsibilities, each of which is represented in the framework by an abstract interface illustrated via the inner octagon in Figure 1. The concrete implementation, and thus the run-time behaviour, of each subsystem may be provided by the user in the form of a dynamically linked library or through the use of our pre-supplied default concrete implementations as shown by the outer octagon in Figure 1. During application execution, the central framework hub performs dynamic allocation and binding of sub-system implementations to their respective interfaces; the hub also acts as an intermediating interface between the various subsystems.

A number of concepts are used throughout the application framework in order to increase its extensibility and flexibility, and these form a basis upon which further functionality can be built.

- *Plug-ins* A plug-in is a portion of code that is compiled into a dynamically linked library file, commonly extending a predefined interface that is exposed by the application code. At runtime, each plug-in is bound to the application, and is then able to exhibit its contained runtime behaviour via the predefined interface.
- *Data driven programming (DDP)* In traditional object oriented programming, objects are described using classes, which define the state and behaviour of the modelled real-world object. Inheritance hierarchies are used to organise objects with shared state or behaviour. In DDP, state and behaviour are described separately from their owning objects as components, reflecting a favouring of aggregation over inheritance [6]. By using data to describe component parameters and combinations, a class hierarchy can be defined using one or more data files.

As illustrated in Figure 1, each subsystem in the application framework is accessed via its framework interface. Providing they adhere to the contract described by the subsystem interface, users can replace the default behaviour of

most application subsystems with their own tailored implementation. Through a combination of subsystem specialisations, users can take advantage of the framework's flexibility and modularity in order to build a series of very different applications. For example, a user could reduce the complexity of an applications's visualisation, and instead provide additional functionality for data input and analysis.

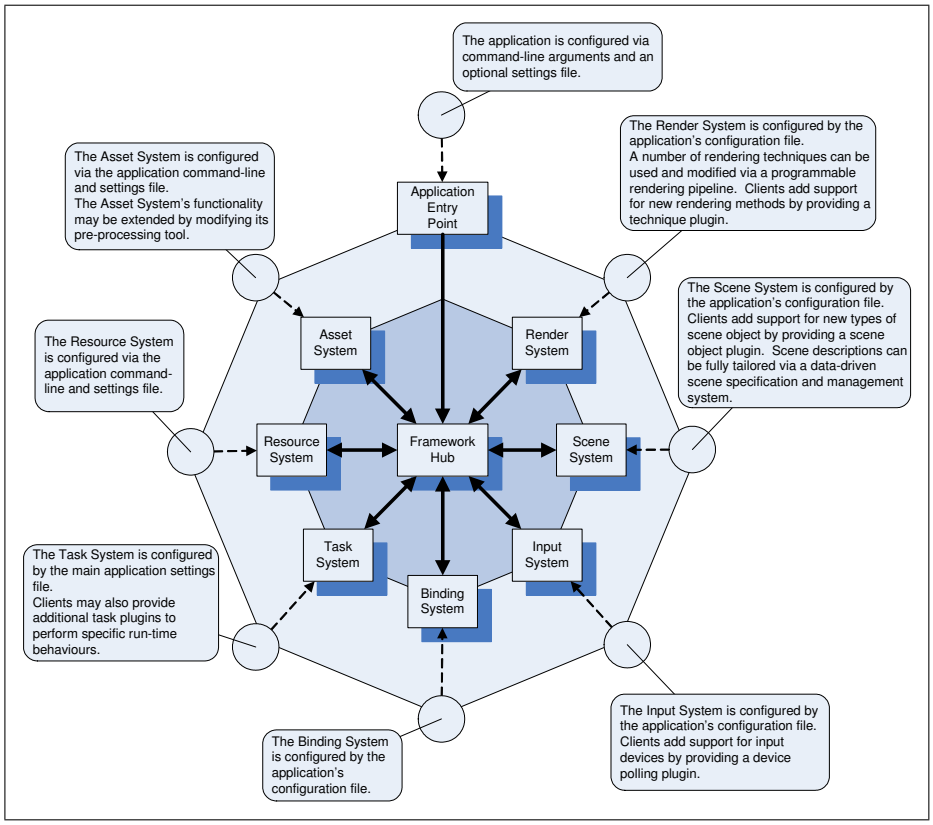

Fig. 1. An overview of the application framework showing the core systems.

2.1 The Data Pipeline

Figure 2 shows the data processing pipeline represented by the framework's data preprocessing tool, asset system and resource system. The framework's data pipeline makes use of a fixed enumeration of asset types, each of which corresponds to an intended mode of data usage, as shown by Table 1.

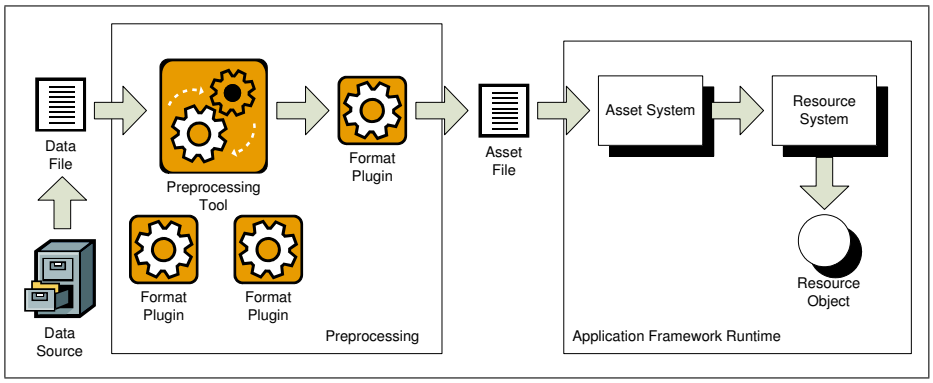

Fig. 2. An illustration of the data processing pipeline. Note that the preprocessing tool maintains a collection of format plug-ins, from which it selects an appropriate plug-in to process any given data file.

Asset Type	**Description**
Text Asset	Text assets represent a contiguous block of immutable textual characters; as such, they are the most fundamental of application asset types. Anticipated uses of the text asset include documentation and fixed-length string storage, for example the user may wish to store interpreted-language AI scripts or user instructions as text assets.
Tree Asset	Tree assets correspond to a tree of named nodes, each of which may contain zero or more named attribute values of a supported type. Example uses of the configuration asset type include the definition of tree-like run-time structures such as scene graphs, and the storage of hierarchical data such as object and property inheritance trees.
Array Asset	Array assets are intended to store multi-dimensional arrays of values of a range of data types. Example uses of the array asset data format include the storage of n-dimensional tables, and n-dimensional textures.
Mesh Asset	Mesh assets correspond to a collection of 3D vertices alongside a system of specifying interconnection based on sequences of vertex indices. The mesh asset format will support a variety of 3D concepts, including animated 3D models with texture coordinates and volumes, such as bounding volumes.

Table 1. Asset Types

Preprocessing Tool

The preprocessing tool embodies a conversion process, where one or more source files may be compressed and encrypted, and are ultimately written as one or more binary files whose format correspond to the framework's asset types. The behaviour of the preprocessing tool is driven by a combination of command-line arguments plus an optional configuration file, and can be extended through the use of format plug-ins. Each format plug-in represents the conversion process from source data to asset data for a single source data format. The use of format plug-ins results in a preprocessing tool that supports a diverse, extensible range of input formats, providing their data corresponds to one or more framework asset types.

To continue the running example presented in Section 1, the preprocessing tool will be responsible for converting the traffic simulation's ITN, topography and other data into their corresponding framework asset file types. We will assume that the ITN data is described using Geography Markup Language (GML); the user must either obtain or develop a format plug-in that can validate and convert the ITN data to the application framework's array asset format. In this case, we choose to represent the ITN as a two-dimensional table that maintains the original's topology information. During the preprocessing tool's execution, the specified ITN files will be read and their data processed by the assigned plug-in(s), which will in turn output one or more array asset files to be read by the framework's asset system. The traffic simulation system's other data files will be similarly processed according to the user's configuration.

Asset System

The asset system represents a repository of asset files, and is responsible for maintaining this collection and providing efficient access to its data. The asset system is therefore synonymous to a file system, albeit one with a fixed range of file types. For example, a given implementation may represent a locally stored directory tree of asset files, a networked or ftp-based cache of asset files, or a web (service) based catalogue of compressed and encrypted asset archives.

In the context of our running example, the traffic simulation's data may be distributed as a number of compressed archive files. The ITN array asset produced above, plus a number of other representations of the same road network (for example, the roads' geometry in the form of mesh assets), are supplied as a single archive file. Further archive files contain data-driven descriptions for the various vehicles that will populate the simulation. A final archive file contains the scene descriptions and application configuration files in the form of one or more tree assets. The asset system implementation will be responsible for locating a given asset file within these archives, and providing access to asset data when required.

Resource System

The framework's resource system is responsible for maintaining a collection of run-time objects, each of which represents the data held by a single asset file. While the asset system provides low-level access to asset data, the resource system's framework interface requires that any given implementation is capable of mapping an asset identification string (such as a file path or URI) to a run-time object that provides the corresponding asset type's modus operandi. For example, resources can be constructed from asset data in a background thread in order to hide load-time delays from the user or resources can be incrementally or partially constructed according to the application's data requirements, e.g. in level of detail implementations.

The traffic simulation's resource system implementation will build runtime objects that allow the data to be used in a meaningful way: tree assets will be represented as hierarchical data structures, array assets will be represented as N-dimensional arrays of data items of a described type, and so on. The default implementation constructs such resources in a background thread.

Summary of the Data Pipeline

The data pipeline described here represents an optimised route for static file-based data. A data pipeline for more dynamic data, such as streaming input, and data that is not file-based, such as web (service) content, is realised by a combination of the input and binding systems (see Section 2.2).

The definition of a fixed range of asset types results in a predetermined format for data manipulation, which in turn allows data processing (that is, parsing and validation) to be reassigned to an offline stage. A fixed range of asset types also aids the design of a concrete asset system interface, which allows the details of asset collection and access to be decoupled from the application framework's other responsibilities. The resulting data pipeline, shown in Figure 2, can be optimised for efficient throughput of a known range of data formats.

The traffic simulation example demonstrates how the data processing pipeline can be tailored in order to support a given use of data. The preprocessing tool has been extended to support a variety of input files, and the asset system has been specialised to support a chosen asset distribution scheme.

2.2 Application Kernel

The application kernel represents the processing heart of the application framework. The task system encapsulates application behaviour and functionality, while the binding system allows subsystem implementations to communicate effectively and store arbitrary data in a type-safe, centrally controlled manner. Together, these two core subsystems provide a backbone of functionality that forms the basis of further application behaviour.

Task System

At a high level of abstraction, the task system's overall behaviour takes the form of iteration over a number of distinct time slices, each consisting of a number of subsystem operations or events occurring in a given order; this is illustrated by Figure 3. The task system thus represents what is traditionally termed an *application loop*. Tasks submitted to the task system are ordered and subsequently triggered according to their priority value, which is provided by the submitter. When triggered, a task is supplied with a summary of the task system's status, along with access to the framework hub and hence the state of the application framework as a whole. The task objects themselves are both defined and supplied by subsystem implementations or as task plug-ins.

Each task plug-in provides a single task object to be submitted to, and thus processed by, the task system. Task plug-ins represent one way in which users can extend existing framework functionality, by providing additional behaviour to be exhibited at run-time. For example, a user could write and submit a task plug-in that regularly monitors congestion levels along a number of inner city roads. Similarly, another task plug-in could randomly dispatch emergency response vehicles in order to test alternate routes through the traffic network.

Tailored implementations of the task system can take advantage of dual core processors or distribute available tasks over a number of clustered machines, and can thus represent a customised task scheduling and distribution policy. A user in need of greater flexibility could extend the task plug-in concept in order to expose application framework functionality to scripting languages such as Python[8] and LUA[9].

Binding System

The framework's binding system is a repository for run-time data of any type. Data is associated with an identifier, and is stored as part of a hierarchical collection of *namespaces*. Subsequent to storage, data can be accessed through the use of a type-safe binding object. While subsystem interfaces present a fixed channel for inter-system communication, the binding system can be used for more implementation dependent storage and interaction.

To illustrate binding system use, a summary of the traffic system's current state can be stored as a dedicated compound type bound to an appropriate location in the binding system. Specialisations of one or more application subsystems, or alternatively application plug-ins, could then bind to and use this information in order to affect task scheduling, select resource construction policies, or inform the user as part of a graphical user interface (GUI). The

[8] `http://www.python.org/`
[9] `http://www.lua.org/`

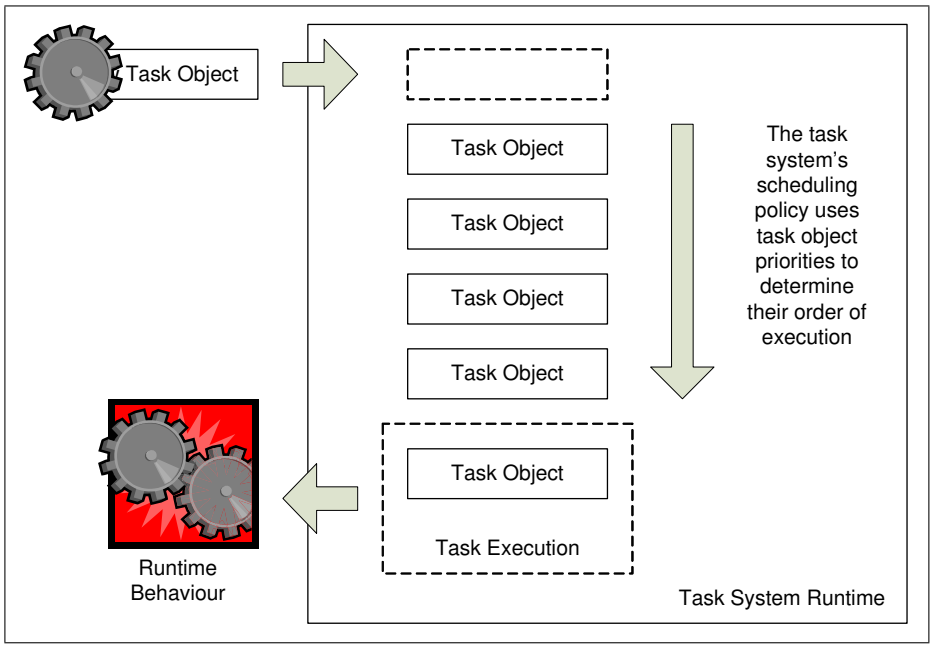

Fig. 3. An overview of the task system's functionality.

traffic system's summary information could also be fed back into the simulation itself, so that emergency vehicles avoid congested areas and vehicles choose alternate routes to avoid icy roads.

Input System

The input system is a logical extension of the binding system; it represents a collection of objects which periodically update data bindings. Input plug-ins, each representing a single data source, are registered with the input system, which polls each data source during the main application loop. Typically, data sources will be input devices such as the mouse or keyboard, but these could also include other sources such as database or internet connections, value generators, procedural models, and so on. This is in contrast to the data processing pipeline presented above, which is intended to provide access to static data sources, and does not support dynamic streaming content by default.

An input plug-in can easily represent a device such as a mouse or keyboard. An input plug-in, along with its associated data bindings, could also represent a more complicated model, such as a numerical weather prediction system. In this case, the input plug-in may be connected to an online database providing current or forecast weather data. In the context of the traffic simulation

example, the weather data stored by the binding system could be accessed by other parts of the application framework and adjust vehicle spacing due to altered stopping distances and visibility or increase the probability of an accident occurring in icy conditions.

Summary of the Application Kernel

The application kernel forms a collection of low-level functionality upon which further developments can be made. The task system provides an abstraction of the application loop, represents a systematic processing of runtime behaviour, and allows users to inject additional behaviour where required. At runtime, certain tasks may be polling a diverse range of input devices and data sources, and writing values to bound variables. Other tasks may be querying the value of variables that have been identified by name via the binding system's interface.

2.3 Simulation and Visualisation Components

The subsystems described here build upon the functionality provided by the lower level framework subsystems in order to support the simulation and visualisation of many different spatiotemporal environments. While the scene system maintains the topological and spatial representations of a given environment, the render system makes use of modern developments in rendering technologies in order to present a powerful yet flexible visualisation pipeline.

Scene System

The scene system is responsible for maintaining the runtime state and content of a given spatiotemporal simulation. The scene system maintains two representations of the simulation environment and its constituent objects: a spatial partitioning system to maintain the spatial relationships between objects, and a scene graph to embody the high level topological aspect of the environment, which are shown in Figure 4. While the former representation will allow for efficient spatial queries such as proximity and collision detection, the latter representation makes heavy use of data driven programming (DDP), which introduces a further aspect of extensibility and flexibility to the framework's overall design. The application framework's scene system will use data driven objects to populate its simulated environments, which means that users will not only be able to stipulate scene composition using data, but will also be able to describe new types, and extend the definition of existing ones, via data manipulation.

The scene system describes objects as a composition of object components, or *facets*, as described in Table 2. Additional facet types may be supplied as *facet plug-ins*, which define subtypes of those presented in Table 2, allowing

users to identify new ways in which to describe scene objects without having to develop or modify the scene system implementation. For example, a behaviour facet plug-in could allow Python scripts to stipulate the runtime behaviour of vehicles in the traffic simulation.

In practice, scene object *types* will be defined by the user using XML. The object type definition will include an element for each facet that contributes towards the object's functionality, and facet plug-in developers will typically provide XML schema that can be used to validate descriptions for their facet types.

Object types can also form part of a data-driven object hierarchy, through the use of object type *inheritance*. When defining a new object type, users can also specify that the new object type is a subtype of an existing parent object type. Conforming to traditional object oriented software concepts, child object types *inherit* or *override* properties of their parent types. The inheritance scheme described here is applied at the facet level, so a child type description is free to override some behavioural parameters while inheriting others.

The benefits of DDP are increased extensibility and flexibility; the definition of new object types and behaviours when linked to a scripting language, as well as the modification and instantiation of existing ones, can all be achieved via data manipulation *without access to application code*. In the context of visualising GIS information, the use of DPP enables the user to assemble information rich virtual environments through the combination and extension of existing scene object type descriptions.

Render System

The render system is responsible for the visualisation of the spatiotemporal simulation. While geometry and texture properties are supported by the framework's various asset and resource types, appearance properties are described using nVidia's Cg language and the CgFx effect framework.

Recent developments in graphics hardware are now able to bring the rendering capabilities of even basic machines close to that of dedicated systems. While past incarnations of both hardware and software APIs have utilised a fixed functionality pipeline for transform and rasterisation, today's hardware and software interfaces support a flexible programmable pipeline that exposes key functionality to the client. Programs written in a dedicated language, known as *shaders*, stipulate the appearance of objects in a given virtual scene by specifying light, material, and surface characteristics alongside scene-wide effects such as shadows.

The rendering subsystem is based around the use of nVidia's CgFx effect framework [7], that aims to maximise flexibility without sacrificing runtime efficiency or ease of use. Improving image quality in the context of 3D visualisation traditionally requires additional detail and accuracy, which in turn translates to increased geometrical and computational overheads, thus reducing application response and user interactivity [8]. A better method of

Facet Type	Description
Data Facet	A data facet represents a collection of named variables whose initial values may be specified as part of a scene object description. For example, a car object may have an engine size, fuel level and registration associated with it.
Behaviour Facet	A behaviour facet's functionality is similar to that of a task plug-in, although a behaviour facet also has access to the instance to which it belongs, along with that instance's constituent facets.
Bounding Volume Facet	A bounding volume facet simply specifies the spatial boundary of its owning scene object. For example, the bounding volume of a vehicle may be defined as an axis-aligned bounding box.
Scene Graph Facet	A scene graph facet represents its owning object's node in the scene system's hierarchical representation of the simulated environment.
Geometry Facet	A geometry facet stores the geometry associated with a given scene object, although this will typically take the form of a reference to a mesh asset file.
Appearance Facet	An appearance facet is used to determine visual appearance of its owning scene object. A scene object's appearance is described using nVidia's CgFx format (see later). For example, the appearance facet of a car object type may provide a CgFx fragment alongside default colour parameters that together give all cars a glossy gray appearance.

Table 2. Scene Object Facet Types

improving the level of visual realism would be to focus on image-based techniques like bump mapping and shadow mapping [9, 10]; such techniques have demonstrated that additional detail can be produced via attribute maps and vertex and fragment manipulation [11].

Summary of the Simulation and Visualisation Components

The scene and render systems together form a modelling and visualisation environment that is capable of supporting a wide range of applications. The data driven implementation means the user is thus able to influence the modelling of a given simulation, and its visual output, via text-based modifications. In the traffic simulation example, users can describe a basic car type though a combination of facet parameterisations using XML. Within the render system users can provide a catalogue of vehicle geometry files and material properties, which are used in various combinations to illustrate an assortment of different vehicle types and colour schemes.

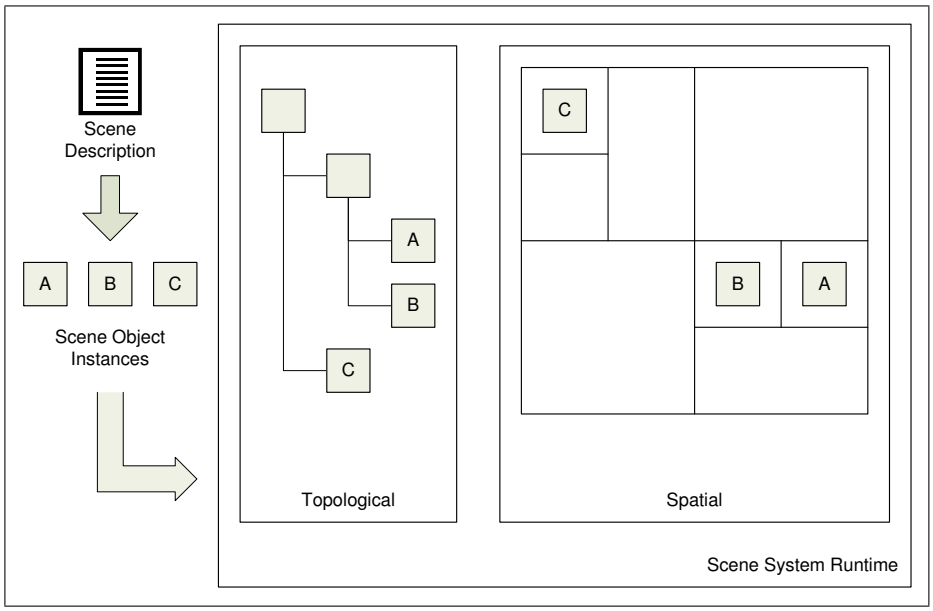

Fig. 4. An overview of the scene system's organisation.

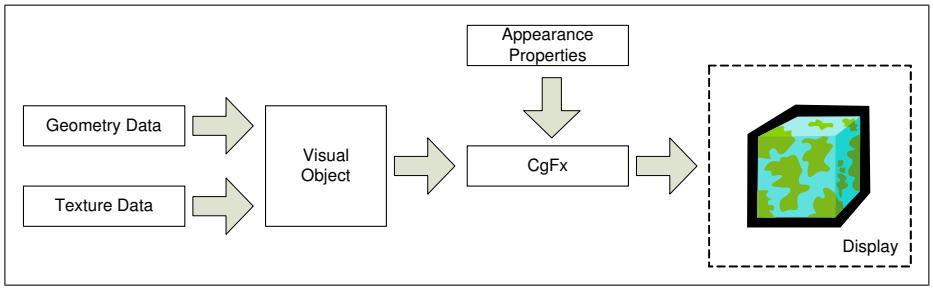

Fig. 5. An overview of the render system's functionality.

2.4 Summary of the Running Example Application

Examples throughout this paper have demonstrated how elements of a real-time traffic simulation system could be implemented using specialisations of, and extensions to, the application framework.

Section 2.1 explains how the application framework's data pipeline can be modified in order to support a given input format and distribution method. The modifications allow the traffic simulation to use a variety of data formats, including a GML based data. A specialisation of the asset system allows the

application framework to locate and access the resulting asset data, which provide application content and are used to drive runtime behaviour.

Section 2.2 describes a number of alterations that allow users to define application behaviour. Further examples show how the binding and input systems can be used to obtain and store data from real-time sources, such as a numerical weather prediction database, and use this data to influence the traffic simulation.

Section 2.3 gives examples of how a data-driven object and scene description system can be used to provide users with a flexible, extensible tool for defining the state, behaviour and appearance of runtime objects.

3 Conclusion

In this paper we have shown the framework for a novel visualisation system we are developing. Central to the design of the framework is careful attention to the ease with which the application can be extended or modified to suit particular visualisation tasks. Through the design shown above we have been able to ensure that almost all parts of the system can be modified or extended, some using plug-ins, others through a data driven approach, including the use of fast scripting languages. Our aim is to create an open source base platform that can be extended by us, other members of the visualisation programming community, or users of the system to address a range of requirements. More fundamentally we expect that a range of plug-ins for import of a range of data formats will be created, and possibly a range of plug-ins for driving specialist visualisation hardware. The careful design of the system means that this can be achieved easily without any need to recompile or, for the data driven aspects, even code.

In future work we are looking at extending the application framework to add GIS functionality to create an integrated modelling and visualisation package. We are also exploring the links that we can usefully make between GML3.1 and the data driven components in the scene system.

References

1. A. Herwig and P. Paar, *Game Engines: Tools for Landscape Visualization and Planning?*, pp. 162–171. Wichmann, 2002.
2. D. Fritsch and M. Kada, "Visualisation using game engines," in *Geo-Informations-Systeme*, vol. 2004, pp. 32–36, June 2004.
3. B. Kot, B. Wuensche, J. Grundy, and J. Hosking, "Information visualisation utilising 3d computer game engines case study: a source code comprehension tool," in *CHINZ '05: Proceedings of the 6th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction*, (New York, NY, USA), pp. 53–60, ACM Press, 2005.
4. S. Patterson, "An object–composition game framework," in *Game Programming Gems 3* (D. Treglia, ed.), ch. 1.2, pp. 15–25, Charles River Media, July 2002.
5. K. Appleton, A. Lovett, G. Snnenberg, and T. Dockerty, "Rural landscape visualisation from gis databases: a comparison of approaches, options and problems," *Computers, Environment and Urban Systems*, vol. 26, pp. 141–162, 2002.
6. A. Shalloway and J. R. Trott, *Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition*, p. 429. Addison-Wesley, London, 2005.
7. R. Fernando and M. J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.
8. M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar, "Interactive multi-pass programmable shading," in *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 425–432, ACM Press/Addison-Wesley Publishing Co., 2000.
9. J. Wang and J. Sun, "Real-time bump mapped texture shading based-on hardware acceleration," in *VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry*, (New York, NY, USA), pp. 206–209, ACM Press, 2004.
10. M. Stamminger and G. Drettakis, "Perspective shadow maps," in *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 557–562, ACM Press, 2002.
11. A. J. Claude and M. Stevens, "Leveraging high-quality software rendering effects in real-time applications," in *GPU Gems* (R. Fernando, ed.), ch. 35, pp. 581–599, Boston, MA: Addison Wesley, 1 ed., 2004.