Chapter 5

Cadence Models and Examples

The Cadence prototype described in the previous chapter is best illustrated and evaluated by developing a variety of models within it. Over the course of this project many models have been developed by several people and it is the purpose of this chapter to go through a few of these. Each model is used to demonstrate some characteristic of Cadence that is relevant to future discussions and to show how it relates back to the objectives of the work. Further details of the models discussed here are given in Appendix ???. It should be remembered that the main comparison is with existing EM tools although some aspects demonstrated by these examples are novel in their own right and go beyond EM concerns.

5.1 Stargate

There are an increasing number of approaches to composing programs in a flexible way, usually involving XML to link certain components at run-time rather than embedding these structural dependencies into the source code. This is known as dependency-injection [ref]. Cadence can be used for this purpose and the best way to show this is to look at how the previously introduced Warwick Game Design library is used in Cadence. The game library consists of many different C++ classes to provide specific functionality

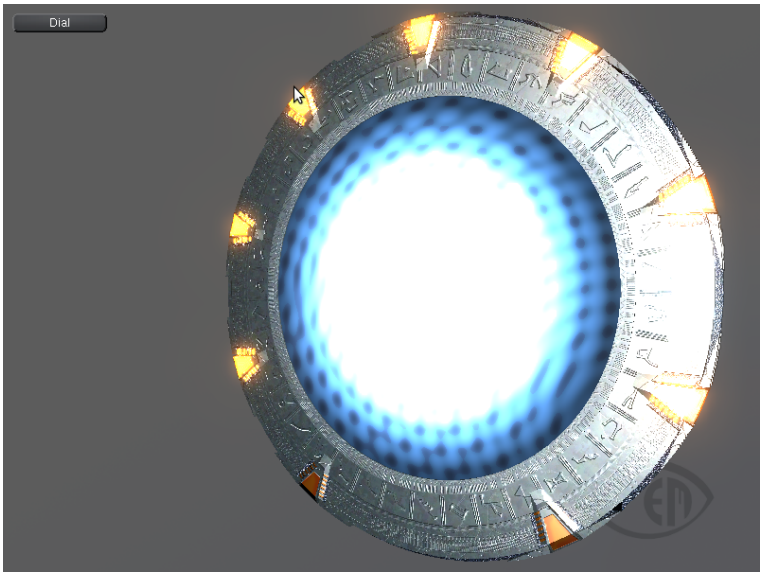

Figure 5.1: The Stargate model

such as loading textures into memory ready for drawing on the screen using OpenGL. These classes have been kept as independent as possible from each other in code. As a consequence they can be combined in interesting ways at run-time as part of a model. Each component has properties that are visible to the Cadence environment and can be given dependencies and values to link them into models. To illustrate this point a model of the Stargate is used which makes extensive use of the game library and glues a huge number of components together to produce all the visual effects.

The Stargate is an artefact from several films and a television series that generates a wormhole between planets for fast transportation [ref]. Originally the model was developed to test out the graphics capabilities of the game library whilst also working out the visual and animation details for a game a student member of the Warwick Game Design society had wanted to make involving a Stargate.

To generate a model as visually complex as the Stargate involves many components including textures, materials, geometry, pixel shaders, cameras, light sources and a scene manager. In addition to this is the widget system that allows the whole scene

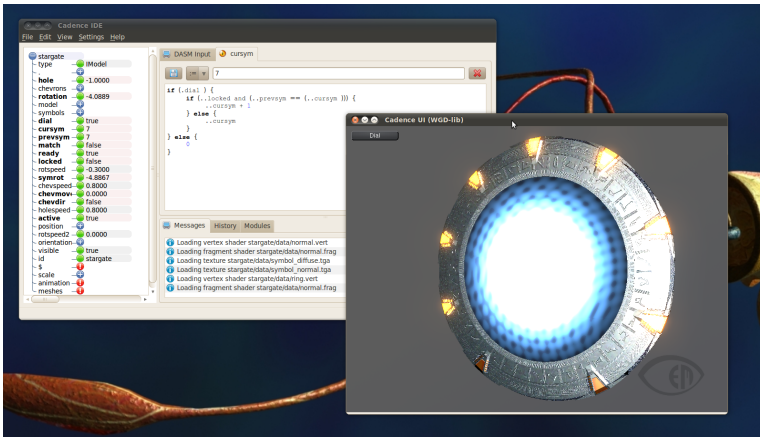

Figure 5.2: Stargate model with the Cadence IDE showing a selection of the Stargate observables

to be drawn within a moveable, resizable window along with any other models. The structure that results from combining all of this has over 200 components just for the Stargate alone. A diagram is given in figure ??? to show how these components relate. The diagram only shows the most significant components and only the structural relationships, not the logical dependencies that may also exist. Note that nodes shown in grey indicate that a C++ game library agent is associated with that object to generate OpenGL commands or provide some other functionality based upon the properties of the Cadence node. The highlighted region is expanded in figure ??? to give an idea of the logical dependencies that also exist between components. Both diagrams could conceivably be automatically generated from information contained within the model, although in this case they have been drawn manually. In existing EM tools there is support for the drawing of dependency diagrams [ref].

As an example of how Cadence is used as glue lets focus on the shaders that are combined to produce a HDR (High-Dynamic Range) bloom effect [ref]. Figure ??? shows a more detailed view of this part of the model. The effect is achieved by first drawing the whole scene into a buffer in memory and then using that as a texture for another rendering. This process is repeated many times and each time the scene is

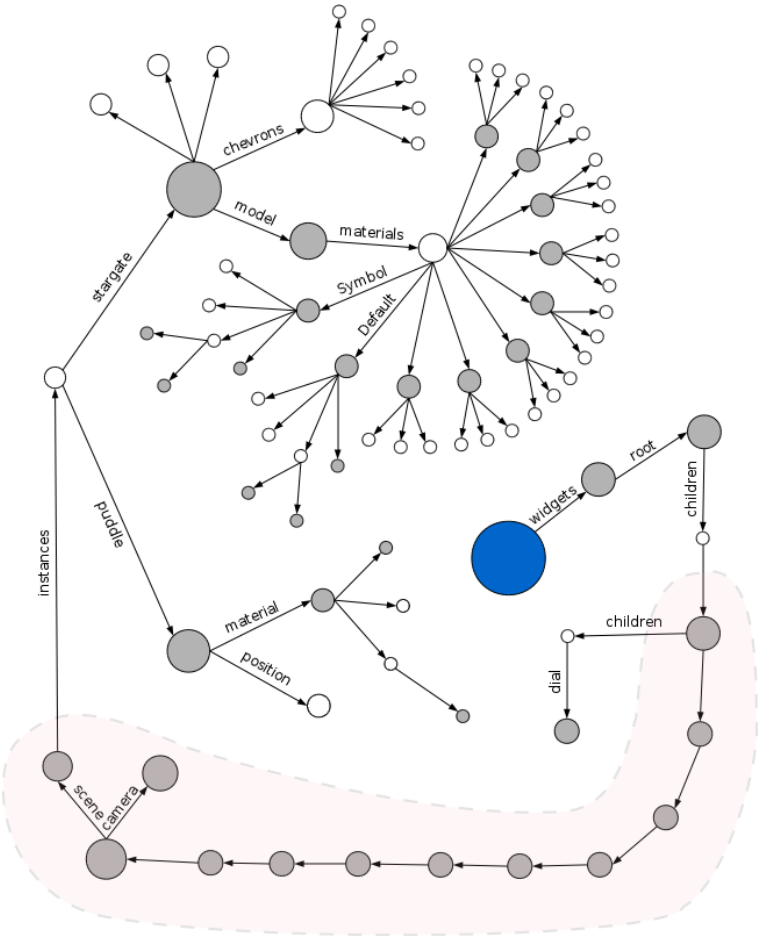

Figure 5.3: Graph structure of the Stargate model. Grey nodes indicate the existance of a C++ game library agent attached to that object. The black node is the root node.

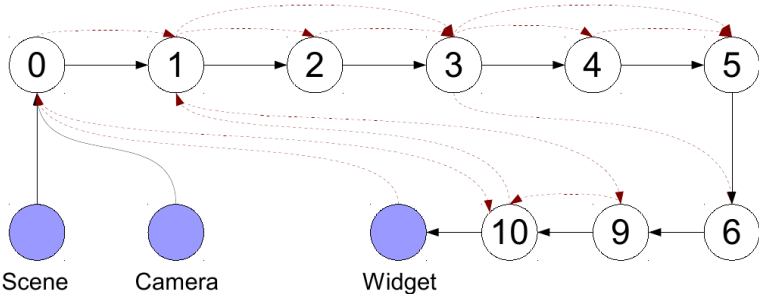

Figure 5.4: Stargate bloom component structural and logical dependencies

rendered at a different resolution or with a different pixel shader. Cadence allows these buffers, textures and shaders to be combined in custom ways and this can be changed whilst the model is running to develop new effects. Properties of the components are connected by dependencies, for example the size of one frame buffer needs to be proportional to another one so a formula is written to describe this. The formula can be written as follows and corresponds to one of the red dashed dependencies in figure ???:

```
width is { @bloom 1 texture width / 4 }
```

Ultimately the size goes back to the screen resolution being used so if this changes then all buffers are updated automatically and atomically by Cadence. Such dependencies exist all over the Stargate model but once written they can generally be forgotten.

As well as the glue aspect of Cadence the Stargate model illustrates the use of temporal dependencies for animation purposes, something not possible in any existing EM tools. These temporal dependencies have been used to rotate the dialing wheel and move the chevrons. This combined with additional dependencies describing the overall state of the Stargate enables an animation sequence to be developed in what is hoped to be an intuitive way. Such use of temporal dependency goes beyond simple animation and infact becomes a fundamental part of the model itself since the logic of the dialing sequence depends upon the rotation. Here is an example of a temporal dependency being used to rotate the inner dialing wheel of the stargate:

```
rotation = 0.0
rotation := {
    if (.dial and (.match not or (.locked)) and (.ready not)) {
        ..rotation + (..rotspeed * (@root itime))
    } else {
        ..rotation
    }
}
```

As can be seen, the definition makes use of various other observables in the condition to decide whether or not it should be rotating. One such observable is *match* and is itself defined as follows:

```
match is { .rotation < (.symrot + 0.1) and
    (.rotation > (.symrot - 0.1)) and (.dial) }
```

This *match* definition shows how the logic of the model depends upon the rotation, in this case to check if the dial wheel is lined up with the symbol it is trying to dial.

Besides showing the graphics capabilities, testing how well the glue like system worked in that role and the use of temporal dependencies it also gave us an opportunity to work on a real Empirical Modelling style project where the resulting model is not well known. The development of the model was experimental and exploratory in nature so gradually evolved as the modeller grew to understand the relationships present in the model. At first the modeller constructed the visual artefacts and located them correctly in the virtual world. Following this they attempted to get the Stargate moving and the puddle displaying, although these were controlled by the user at this point. After some experimentation to get this right the whole process was gradually automated by adding in more relationships until eventually the user could control the Stargate by dialling an address instead of manually rotating things. During this time the modeller gained a greater understanding of the whole Stargate mechanism as they identified problems and found the correct relationships. Although this is a toy example the same techniques can be applied to more genuine real world problems and as a consequence Empirical Modelling is very useful in an educational setting.

Summary: Such modularity and flexibility to link into more traditional software components is missing in the existing Empirical Modelling tools and is also an example of how EM thinking may actually be applied more widely to make these sorts of problems easier to manage. The use of dependency with a graph structure as a glue between software components is novel and as can been seen with this example is a powerful

and useful concept. This applies beyond just the OpenGL game library components, although that is all that is demonstrated here. Also note that this model has been used for teaching EM with Cadence [ref], something that will be explored further in a later chapter.

5.2 Hardware Device Drivers

A few example models have been developed which explore concepts that potentially relate to operating systems and computer hardware. Considering how EM and their concepts can be applied at a more fundamental system level was an early objective of this work. An early version of Cadence was developed which could run independently on a machine as its own operating system. The intention here was to develop device drivers and all other parts of an operating system within a Cadence like environment with prototype-based objects and dependencies being the fundamental concepts. There are two examples worth including here, 1) a keyboard driver and 2) a terminal output driver. Both illustrate the potential of such an approach as well as the associated problems.

The keyboard driver was successfully developed within the early operating system version of Cadence. It involves reading and writing to IO ports, responding to interrupts and then translating the key code into an ASCII character for further processing later on. All of this was achieved via the dependency mechanism since the IO ports were virtually mapped into the DOSTE graph space, as were the interrupts. For example, when an interrupt occurred it would set a property to true in DOSTE and this would cause other dependency formula to automatically update by checking the values of the IO port properties. When the key code changes as a result of the interrupt it causes the translation mechanism to also update via dependency to change the ASCII character. A simple example but it manages to show how interrupts and dependency mechanisms work well together. The Cadence code for this was originally written in an older syntax, however, below is a small part of the driver updated to the new syntax. The example is for reading the scancode and converting that to an ascii character.

```
.devices ps2keyboard scancode is {
    if (@root interrupts 33) {
        @root io 0x60
    } else {
        0
    }
}

.devices ps2keyboard ascii is {
    if (.scancode < 0x80) {
        .asciimap (.scancode)
    } else {
        '\0'
    }
}
```

A terminal output driver was also developed in the same system so that text could be printed to the screen. This driver is considerably more complex than the above keyboard driver due to the huge number of observables and definitions that would be required if a pure declarative Cadence approach were to be taken. Such a pure approach is in practice impossible as the whole system, including all models, would need to be properly defined. Instead agents need to be used to link parts together since multiple separate models could require output in what ultimately amounts to an unpredictable way. For this reason the old code behind this driver exploited an undesirable feature where definitions could have side-effects. In the most recent version of Cadence this ability has been removed and replaced by an alternative agent mechanism.

5.3 Network Distribution for a Video Wall

Network distribution was another early route taken by the work that has been dropped to focus on other aspects. The idea here was to have the underlying DOSTE graph transparently shared between many machines so that hardware and software could be

accessed as if it were local. The underlying architecture of Cadence supports extensions in a way that enables parts of a graph to be virtual or remote. By sending all events over a network instead of doing local processing it was possible to achieve the desired network transparency at the lowest level. To demonstrate and develop this a video wall was used. This video wall consisted of 15 screens and 8 machines to drive them, each connected over an infiniband network. A 9th machine was used as the server to control the others. Normally a software package called Chromium is used to distribute OpenGL rendering across the machines but this has many limitations in that it only supported a subset of OpenGL functionality at the time. It also sent polygon data over the network rather than any higher-level information about what was being drawn. Cadence takes a different approach. Each machine runs an instance of the Cadence environment but with a slightly different address space for objects and is then connected to a master instance on the master machine. At this point all machines can transparently observe the DOSTE graph of the master machine so when a model is loaded each machine can observe it. It was then a simple task to customise each machine to draw a specific portion of the model from a particular view so that the result appears to be a single view over 15 screens. This customisation could also be done from the master machine.

Performance was adequate despite the network code being poorly written. It had advantages over Chromium in that it could support all OpenGL features available on the graphics cards of the 15 machines. This included shaders. The model used to try this out was the Stargate discussed previously. The Stargate model failed to run over Chromium but worked without alteration in this version of Cadence with all of the visual effects supported. Any model developed within Cadence could be adapted to run in this way and the video wall is just one example for distributed visualisation. Unfortunately the video wall was dismantled before any real data could be collected and analysed. The network code is currently available in the XNet module and there are, at the time of writing, plans to develop this further as an MSc project.

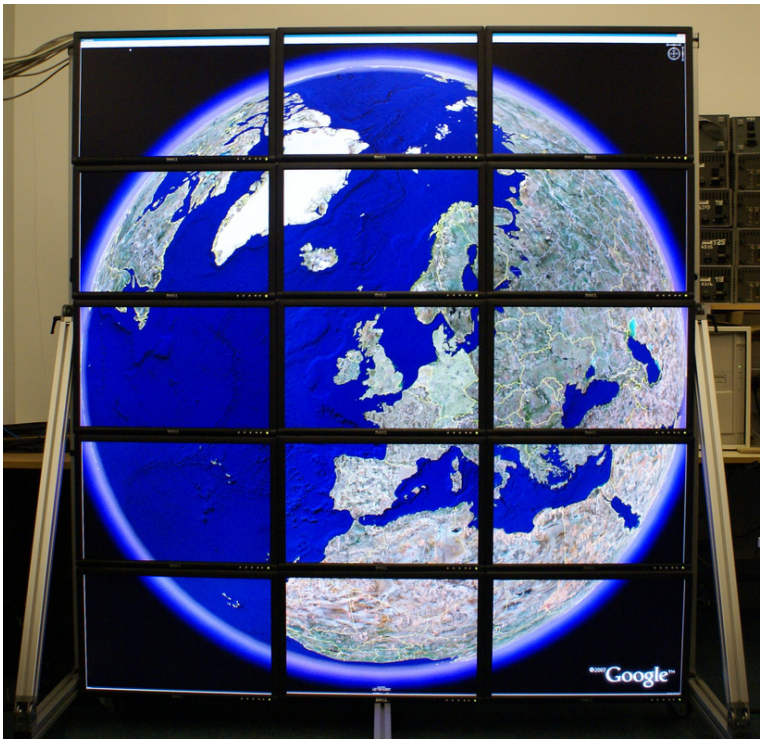

Figure 5.5: Google earth running on the same video wall used for the Cadence work. Photo taken by Richard Cunningham

5.4 Kinesin Biological Model

One aspect of Cadence to be explored is the use of temporal dependencies. There are many models that make use of this concept, mostly for animation purposes, however there is one model that demonstrates the process like nature of these dependencies being applied for more than simply animation. The model is called Kinesin. Kinesin is a motor protein operating within neurons as a means of axonal transport. They move neurotransmitter along microtubule tracks from the cell body to the synapses. The mechanism by which these motor proteins operate is not fully understood and so PhD researcher Richard Wilson has been attempting to construct a discrete dynamical system on a computer to learn about the process [ref]. His attempts have been using the C programming language to experimentally try out different theories he develops. We suggested that perhaps constructing such a model in our tools would provide him with a better platform for such experiments and so such a model was constructed.

The modelling process involved first creating a representation of the motor protein, consisting of two "heads" and a spring-like chain connecting them together. Once the basics were in place some behavioural definitions were added to simulate brownian motion of these "heads". The precise sequence and timing of events is complex but important in understanding how it moves along the tubes. The process involves one head binding to the tube and releasing its ADP with ATP then binding to it causing the spring-like chain to partially attach to the top of the head. This encourages the other head to attach to the tube infront of the already bound head and the process repeats with this head. Meanwhile the first head hydrolyses ATP and ultimately this results in it detaching. By repeating this process the head walks along the tube. As our model progressed we were able to fine tune this sequence, based upon what is known about these various molecules and what was observed in the model, until we successfully and reliably got it walking. During the modelling process we did discover flaws in the original theory and realised that additional processes were involved that had not been considered.

It took considerably less time to reach this point using our tools than it had taken Wilson with his C program. This was due to the ability to observe and modify the process in intuitive ways as it was running, whereas with C it involved stopping the program, changing the source, compiling and restarting. Our model was developed to more closely match the domain by using observables and definitions that had direct meaning rather than dealing with a more abstract machine based approach to state and behaviour. It was the direct, concrete and interactive nature of our tools that proved so beneficial in this kind of work.

5.5 Wii-fly Game

Wii-fly is a game developed for and with the Warwick Game Design society at the University of Warwick. The main purpose was to make a game that used wii remotes. Cadence and the associated game library provided a simple means of connecting wii remote sensors to the controls of a ship. The original and simple version of this game required only just over 100 lines of DASM script to create a world, ship and connect it to the wii remote. From this it was extended, adding menus and several different ships. Eventually some of the code was taken into a C++ agent but it still made use of Cadence.

What this game shows is how a simple module, which uses bluetooth to get Wii remote data, can be plugged into Cadence. This module provides a single agent that sends events into Cadence many times a second to update certain observables corresponding to buttons or accelerometer data. It also shows how Cadence can easily be used to connect this with other observables via various formulas in the form of definitions.

The player of the game controls a ship that flys over a landscape hunting down other players also flying ships. The game can work with up to 4 wii-remotes. During development there was a great deal of experimentation to get the ship behaviour correct as well as the gradual addition of new features. All this was relatively easy because of

the flexible Cadence system lying underneath it all that enabled these experiments to take place without needing to recompile or even re-run the program.

5.6 User Interfaces

5.7 Usability