Seasonal Globe Report

# Design

The following class diagram represents the static structure of the major classes in my implementation of the snow globe application. A brief description of each class concludes this section.



As seen on the diagram, application itself contains one of each “Game Object” (displayed in green), with the exception of particle emitters which application contains two, one for the fire and one for the chimney smoke effects. Additionally each Game Object may contain a number of Graphics Objects (shown in orange). The number of objects varies from class to class with the complexity of their rendering algorithm, for example house needs two textures whereas terrain uses only one.

The following utility classes are also present in the system:



I believe the main merit of my design is the way in which I have wrapped graphics objects. Drawing on previous experience with graphics libraries I believe the design of my graphics objects would allow OpenGL to be swapped with any comparable API such as Direct3D. Swapping graphics APIs is made easier by the fact that no OpenGL calls or data types are used outside of the graphics objects. Other than a dependency on the renderer class (which is providing access to OpenGL extensions), graphics objects have minimal coupling between each other and so I believe are highly reusable.

Wrapping the graphics objects has greatly improved the re-usability of the Game Objects as they do not have dependencies on specific graphics APIs but instead on the services provided by the graphics objects. Game Objects also function independently of one another. While this simplifies the program structure and improves the re-usability of individual Game Objects; it also highlights an area of weakness with this design.

The main problem with structuring the Game Objects this way is the large amount of similar code repeated within each game object. While given the time constraints and simplicity of structure I don’t feel this is a bad solution. If I were to do the project a second time I would have attempted to reduce the repeated code by combining the roles of the game objects into a single class with properties such as in the following diagram:



Implementing this alternate design would be more difficult because the application requires objects to have specific animation events. However, a successful implementation would greatly simplify the code within Application as GameObjects could be entirely specified within the configuration file and iterated over for drawing. My current implementation has each GameObject instantiate its own resources such as textures and vertex buffers, whereas this improved design would require a resource management class provide shared access to them, which would also has the advantage of reducing system and graphics memory when assets are re-used by more than one object.

One area in which I did have time to simplify the code was in the fire and smoke particle systems. Originally system had its own class which shared a large amount of code. I decided it would be better to add a number of parameters to the particle system vertex program and re-use it in a single ParticleSystem class. By making this change late in the project I realised I should have applied that philosophy to the design of all game objects, however it was far too late to consider making such a substantial change. I extended this class to allow efficient batched rendering, as can be seen in the following sequence diagram. Having a single instance of particle systems manage emitters allows the shader to be bound and uniforms updated once for all emitters, reducing unnecessary OpenGL state changes.



## Class Summary

The following table is a complete list of classes present in my application along with brief descriptions, further implementation details may be found within the comment block preceding each class’s header file.

| **Class** | **Description** |
| --- | --- |
| Application | Draws and Updates each object in the scene |
| ArrayElement | Represents the binding of a single Vertex Attribute stream to the provided by a VertexBuffer |
| Base | Handles loading and drawing of the base object which the dome stands on |
| Config | Parses the configuration file and provides properties to an instance of Application |
| Dome | Handles loading and drawing of the class dome object |
| House | Handles loading, drawing and updating of the house object |
| Light | Encapsulates the properties of an individual light which may be provided to a shader supporting lighting |
| Lightning | Handles loading and drawing of a lightning object displayed during winter in the scene |
| LSystem | Provides the capability to iteratively apply a number of rules to an input string |
| Matrix4 | Provides maths functions associated with a 4x4 matrix |
| MyApp, MyWindow | MyWindow is a specialisation of the GLWindowEx provided by GXBase, the specialisation instanciates config and application, and additionally supplies user input to config. MyApp is a specialisation of GXBase’s application class that provides an instance of MyWindow |
| ObjFile | Handles parsing of an “.obj” format model. ObjFile supports automatic generation of index data and interlaced vertex data ready for loading into vertex buffers |
| ParticleEmitter | An instance of a particle fountain |
| ParticleSystem | Provides a mechanism for efficiently drawing a number of ParticleEmitters |
| Pond | Handles loading and drawing of the pond along with generation of the stencil mask for drawing the reflected scene |
| Renderer | Provides a thin wrapper around the OpenGL state. Provides a non-deprecated implementation of lighting, model view projection matrices and the clipping plane. Provides OpenGL extensions to graphics objects |
| Shader, FragmentShader, Vertex Shader | Provides a wrapper around individual GLSL vertex and fragment shaders |
| ShaderProgram | Provides a wrapper around GLSL shader programs |
| SnowDrift | Handles loading, drawing and updating of the elevated snow drift |
| SnowParticles | Handles loading, drawing and updating of the snow particle system |
| Texture | Provides a wrapper around an OpenGL texture object, multi-texturing and loading of texture files from formats supplied by GXBase’s image class |
| Tree | Handles parsing of tree command string, loading, generating, drawing and updating of the procedural tree |
| Uncopyable | Provides a base class specifying a private copy constructor and assignment operator for classes which should not be copied |
| Uniform | Provides a wrapper around the location of a uniform variable so that it may be cached for updating the variable later |
| Vector3, Vector4 | Provides maths functions associated with 3 and 4 dimensional vectors |
| VertexBinding | Provides a wrapper around the vertex binding state, utilises OpenGL’s Vertex Array Objects to provide a mechanism for efficiently re-binding vertex buffers |
| VertexBuffer | Provides a wrapper around OpenGL’s Vertex Buffer Objects which store vertex and index data in graphics memory |

# Graphics Implementation

I have some previous experience with advanced OpenGL, so in this project I wanted to push myself further by choosing to use no deprecated features and stick to those found in the OpenGL 3.0 core profile. By making this decision early in the project I was forced to encapsulate functionality such as vertex buffer objects and shaders in the initial stages of the project. By doing so I had time to explore more effective use of the GPU throughout the project.

## Particle System Implementation

My particle systems are simulated by a vertex program that calculates a unique position for each particle. To render the particles I use GPU instancing to render a quad per particle which is billboarded and transformed by the same vertex program. Each particles position is calculated as a function of its instance ID and time. The instance ID provides a seed which is used to calculate a unique pseudo-random path for each particle. The particle is then offset along this path over time. Particles reset to the start of the path in a fixed time period. This implementation has a number of advantages and disadvantages over a traditional CPU implementation of a particle system.

Particle systems are particularly well suited to the GPU due to its significant parallel processing capabilities. A particle system on the CPU may calculate or update the position of a single particle per core at any point in time and has to send the positions to the GPU every frame. In my implementation the CPU simply has to update an elapsed time variable and dispatch the draw call, while the GPU simultaneously calculates the position of hundreds of particles. This leaves the CPU free to do other intensive calculations which in a game environment may include physics and AI.

By choosing to calculate the particle position as a function of its ID and the elapsed time, the particle system’s memory footprint is reduced tremendously as the position and state of each particle does not need to be stored. While there is plenty of memory available on the lab’s development machines on some systems this may not be the case, especially since graphics memory is often far more limited than main memory.

The main disadvantage of stateless particle effects is the complexity of the implementation, especially for advanced effects. The systems required for this project are simple particle fountains, but other games may need much more complex systems such as swarming or collision detection. I found the transitioning of the particle system between “on” and “off” states to be the most difficult feature to express as a stateless calculation within this project.

A potential disadvantage of my implementation is that the additional calculations placed onto the geometry processors may become the bottleneck on a system with a weak GPU. This wasn’t a problem on the lab machines but it could become a problem for software targeting weaker hardware such as those found in laptop PCs. One way in which this bottleneck could be alleviated, would be in the use of a geometry shader as it would mean that the particle positions are calculated once per particle rather than for each of the 4 vertices of the instanced quad.

## Tree Implementation

I chose to implement the branches of the fractal tree using a lindenmayer system. The L-System rule solver searches a string for instances of a number of patterns. Each pattern has a rule which defines a replacement string. Each character in the output text can then be interpreted as a command for generating geometry. L-Systems are the perfect match for generating trees because they allow rapid prototyping of recursive rules which allowed me to develop realistic trees. I found that to get a natural looking result, randomness needed to be introduced so I extended my rule solver to allow multiple possible substitutions for a matched pattern. Each available substitution was given a weighting so that whenever a pattern was matched a replacement was found through a weighted random selection. I have included a number of configuration files demonstrating various trees, which may be applied by dragging and dropping them onto the executable.

I developed a set of commands for the interpreter that includes rotating, scaling and emitting branches. The commands are listed in full in Appendix A. My interpreter built a list of branches at load time, each branch was represented by a 4x3 transformation matrix of a cylinder mesh. These transformations along with the branches’ level within the tree were packed into a vertex buffer for use during rendering. The rendering process makes use of hardware instancing to render the cylinders with the per instance matrices. The branch growth and death animations are provided by a vertex shader that scaled branches based on their level and a uniform variable that controls the desired draw level.

The branch interpreter has a command for emitting branches that may have leaves attached. The interpreter treats these as ordinary branches but retains a list of them so it can scatter leaves across them. Similarly to branches each leaf is represented by a 4x3 matrix which is packed into an instance buffer for rendering. Leaves have their own vertex shader which calculates a pseudo-random path each leaf to takes when falling from the tree and functions in a similar manner to the particle systems except leaves are rendered as ordinary rotated and scaled quads rather than billboarded.

As the majority of work in generating the tree is done at load time, my implementation requires little more than a single draw call at run-time. When using instanced rendering I found that render time decreased by 2-3 orders of magnitude for a medium to large tree compared to my initial implementation without instancing. This approach once again has the disadvantage that it places additional work onto the GPU so on systems with weaker graphics hardware could become the bottleneck. One way in which the implementation of leaves may be improved could be the introduction of collision detection with the ground. This could be achieved by testing leaf positions against a height map of the terrain.

# Project Management

I divided development into two sections, design and development of graphics objects and design and implementation of features. For the project to be successful it was necessary that the development of graphics objects was completed at an early stage. Development of graphics objects was completed rapidly, although some adjustments, namely introduction of instancing and cached uniform locations, had to be made later for performance reasons. While I tested the graphics objects thoroughly on the lab’s development machines, I much later discovered numerous issues with running my application on NVidia hardware. While I was able to fix all but one outstanding issue (sunlight mode causes the scene to be rendered in black) on NVidia hardware, in hindsight many of the issues would have been far easier to fix if I had tested on more machines earlier in the project.

By completing the graphics objects early, many of the features became significantly easier to implement. Overall some features took more time to implement than expected and others less. For example the lindenmayer system tree took much less time than expected and so I was able to experiment with much more realistic looking trees than those initially planned. In hindsight perhaps some of this time could have been better spent improving other areas of the application as I did allocate several weeks to development of the tree. Other areas such as the implementation of lightning took far longer than expected to create as I was unable to find a suitable mesh for the lightning bolt so had to create my own in a 3D modelling package. In future I will be sure to allow more time for finding, developing and improving art assets.

# Appendix A – Tree Interpreter Commands

The following table contains a complete list of commands available to the tree interpreter:

| **Command** | **Description** |
| --- | --- |
| ‘B’ | Emit a branch at the current orientation relative to the current parent |
| ‘L’ | Emit a branch at the current orientation relative to the current parent which may be the parent of a number of leaves |
| ‘[‘ | Push the current rotation and scale state onto the stack and reset them, use the most recently emitted branch as the parent branch |
| ‘]’ | Pop the current rotation and scale from the stack and step one level up the branch hierarchy |
| ‘<’ | Increase pitch |
| ‘>’ | Decrease pitch |
| ‘^’ | Increase yaw |
| ‘v’ | Decrease yaw |
| ‘+’ | Increase scale |
| ‘-‘ | Decrease scale |