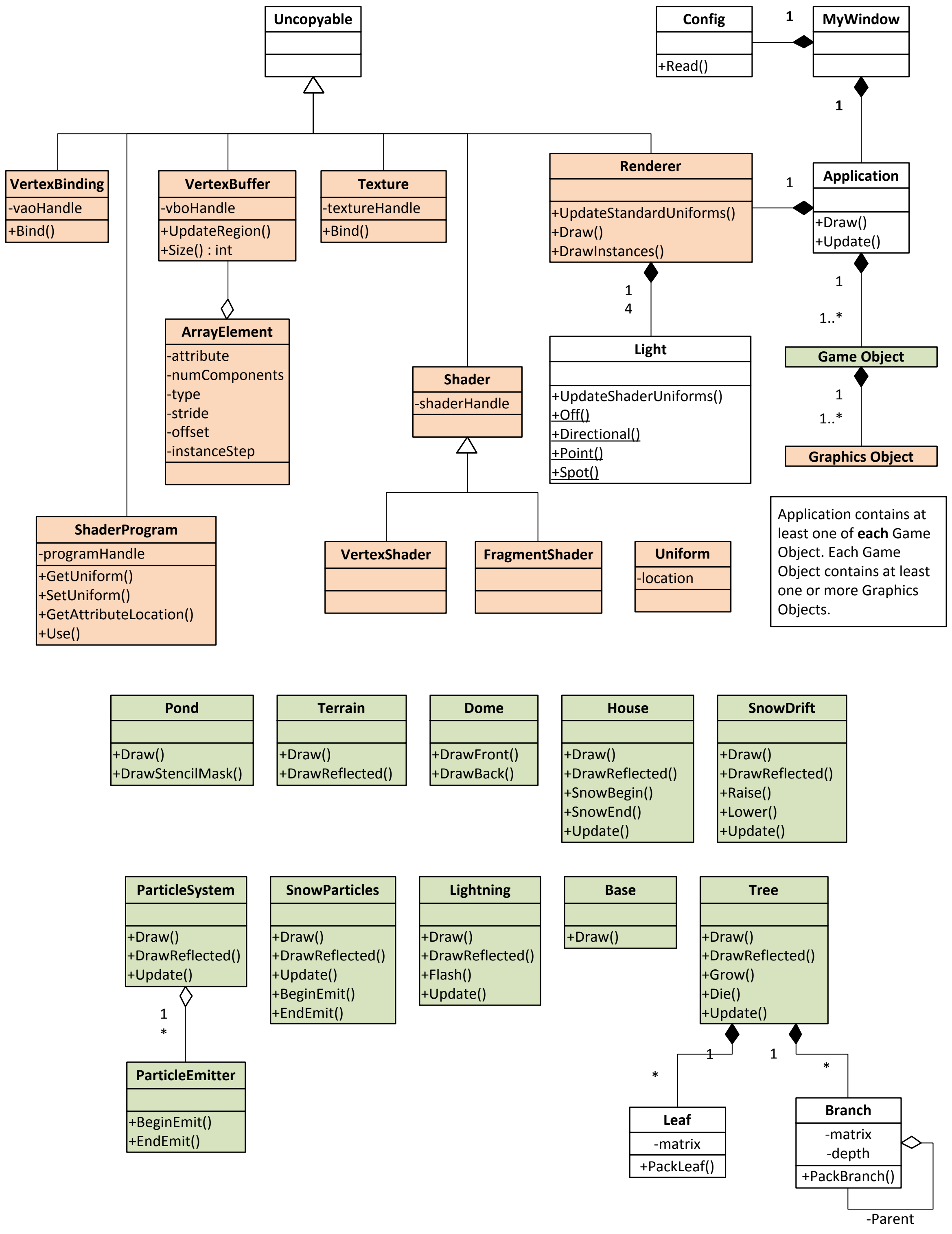
Seasonal Globe Report

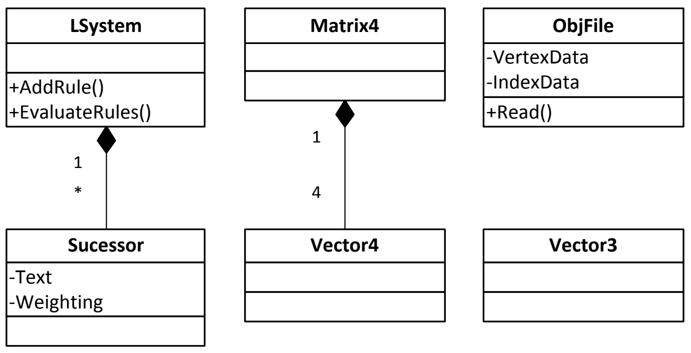
1. Design

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:

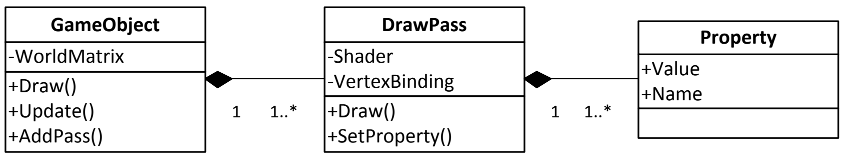


I believe the main merit of my design is the way in which I have wrapped graphics objects. Drawing on previous experience with graphics libraries I believe the design of my graphics objects allows OpenGL to be swapped with any comparable API such as Direct3D. Swapping graphics APIs is made easier by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

Wrapping the graphics objects has greatly improved the re-usability of the Game Objects as they do not depend on OpenGL but instead on the services provided by the graphics objects. Each game object functions completely independently. While this simplifies the program implementation and improves the re-usability of individual Game Objects; it also highlights an area of weakness with this design which is the amount of repeated code.

One area in which I did improve my design was in combining the fire and smoke particle systems into a single class. I later improved the efficiency of this class by separating the particle properties from the renderer, which removes unnecessary OpenGL state changes while rendering. A sequence diagram showing the improvement is included in Appendix A. I was unable to extend these improvements to the snow particle system as the vertex program involved was significantly more complicated and could not be combined without greater overhead.

By changing the particle system late in the project I realised I should have applied the same philosophy to the design of all game objects, however it was far too late to consider making such a substantial change. 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 configurable class such as in the following diagram:



A successful implementation of this alternate design would greatly simplify the code within the application class. GameObjects would be constructed and configured via factory methods and the process of exposing properties to the configuration file would be much simpler. Additionally I would introduce a resource manager class that would offer shared access to resources such as textures and shaders.

1.1.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 to the data 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 point, spot or directional light which may be provided to a shader supporting lighting
Lightning	Handles loading, drawing and updating of a lightning object displayed during winter in the scene
LSystem	Provides the capability to iteratively apply a number of replacement 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,

Class	Description
	that instanciates config and application and additionally supplies user input to Config. MyApp is a specialisation of GXBase’s application class that instanciates 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 of the reflected scene
Renderer	Provides a thin wrapper around the OpenGL state. Provides a non-deprecated implementation of lighting, clipping plane and model, view, projection matrices. 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, multi-texturing and loading of texture files in formats supported by GXBase’s image class
Tree	Handles parsing of tree command string, loading, generating, drawing and updating the procedural tree
Uncopyable	Provides a base class specifying a private copy constructor and assignment operator
Uniform	Provides a wrapper around the location of a uniform variable so that it may be cached for updating the uniform later
Vector3, Vector4	Provides maths functions associated with 3 and 4 dimensional vectors
VertexBinding	Provides a wrapper around the vertex attribute 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

2. 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. This gave me time to explore more effective use of the GPU later in the project.

2.1. Particle System Implementation

My particle systems are simulated by a vertex program that calculates a unique position for each particle and then applies billboarding. To render the particles I render a quad per particle, utilising hardware instancing. 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 pseudo-random path for each particle. The particle is then offset along this path over time. Particles reset to the start of their path after a fixed time period.

Particle systems are particularly well suited to the GPU due to its significant parallel processing capabilities. A particle system on the CPU may calculate the position of a single particle per core at any point in time and would then 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 other game environments 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 doesn't exist in memory. 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 implementation complexity, especially for advanced effects. The systems required for this project are simple particle fountains, but other games may need much more complex systems involving 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 geometry processing 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 mobile chipsets. 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.

2.2. 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 corresponding 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 B. The interpreter builds a list of branches at load time which are represented by a 4x3 transformation matrix, applied to 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 efficiently render all cylinders in a single draw call. The branch growth and death animations are provided by a vertex program that scales branches based on their level within the tree.

The branch interpreter has a command for emitting branches that may have leaves attached. These are treated as ordinary branches but retained in a list which is used for selecting parent branches for leaves. Each leaf is similarly represented by a 4x3 matrix which are packed into a vertex buffer. Leaves have their own vertex program that calculates a pseudo-random path each leaf takes when falling from the tree and functions in a similar manner to the particle systems. Leaves are represented by textured, rotated and scaled quads.

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. By using instanced rendering I found that render time decreased by 2-3 orders of magnitude for a medium to large tree compared with my initial implementation. This approach once again has the disadvantage that it places additional work onto the GPU, which could become the bottleneck on systems with weaker graphics hardware. 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 within the shader.

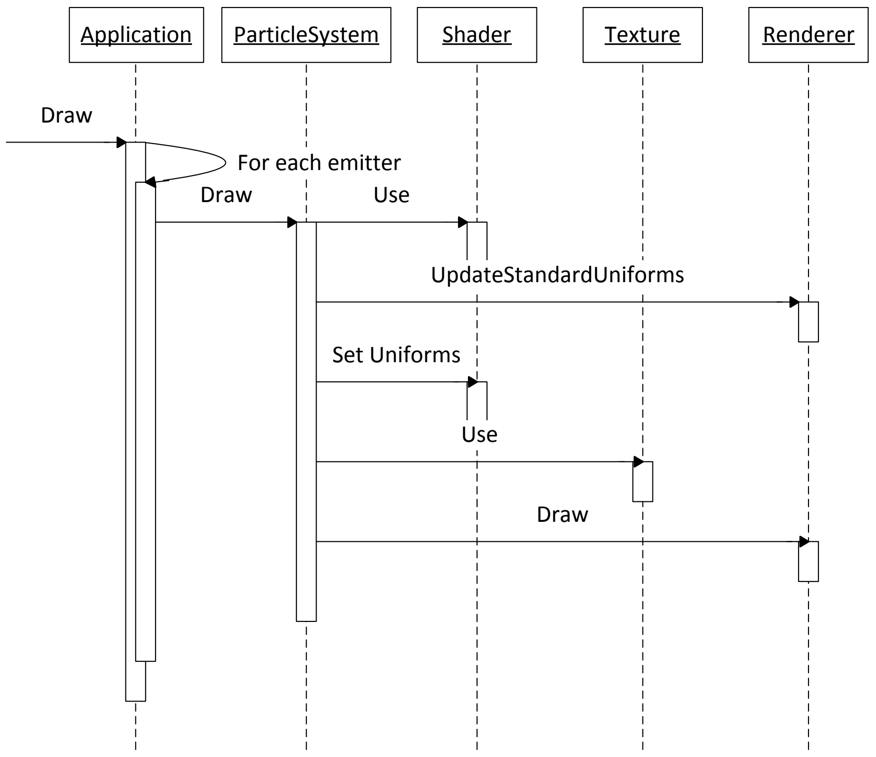
3. Project Management

I divided development into two sections, development of graphics objects and design and implementation of specification 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 were made later for to improve performance. 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 on NVidia hardware (sunlight mode causes the scene to be rendered in black), in hindsight many of the issues would have been far easier to fix if I had tested on more machines earlier in the project.

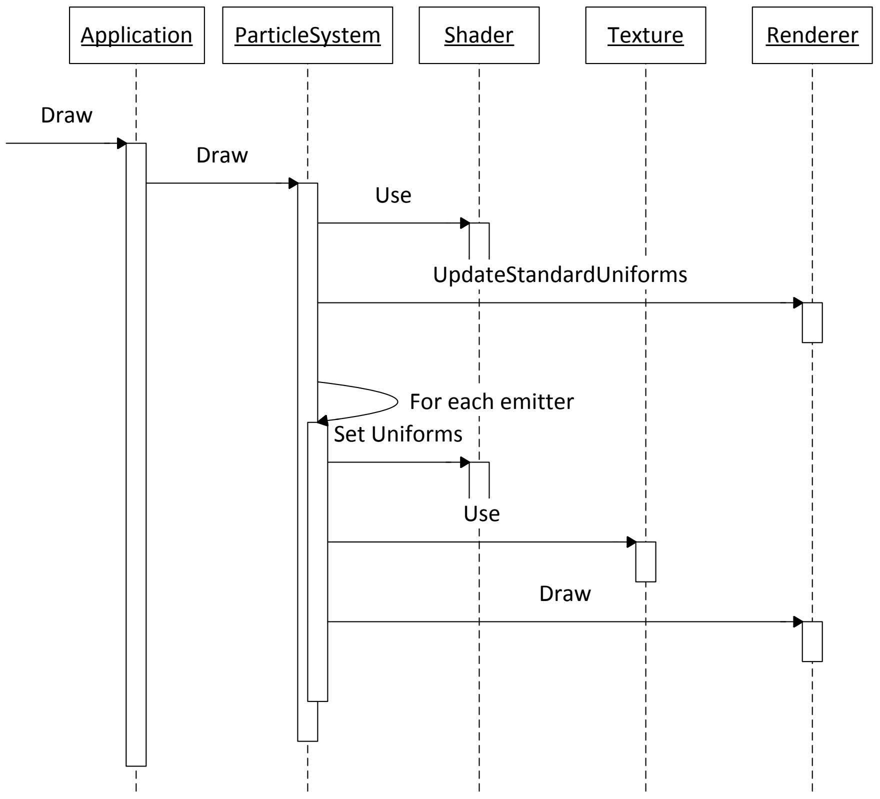
By completing the graphics objects early, many of the features became significantly easier to implement. Overall some features took more time to implement than expected and others less. For example the lindenmayer system tree took much less time than expected and so I was able to experiment with much 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 – Particle System Sequence Diagrams

Particle rendering sequence before optimisations, note that Shader:Use and Renderer:UpdateStandardUniforms are called per effect:



Particle rendering sequence after optimisation:



Appendix B – 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