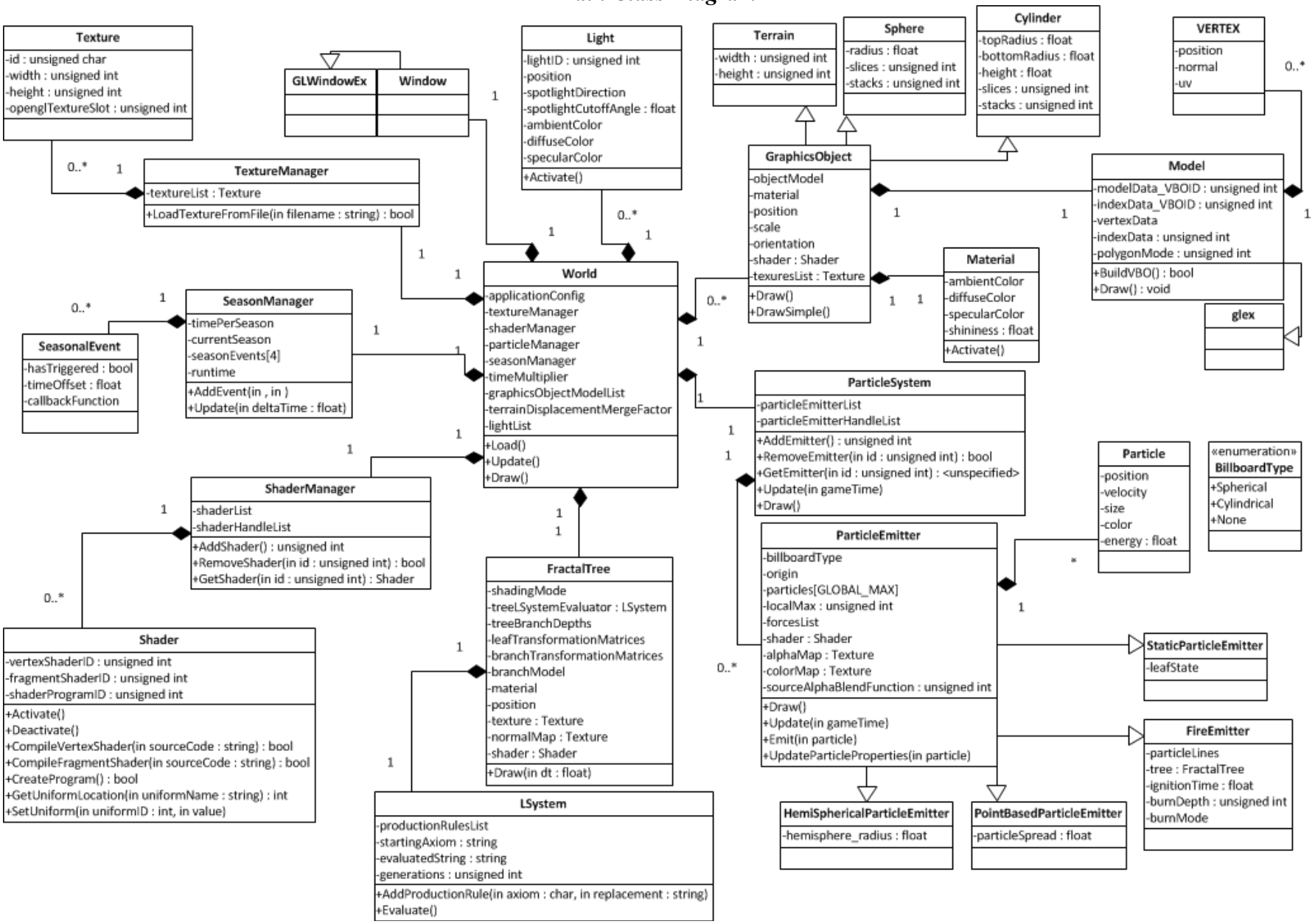
Seasonal Globe Report [currently 2533 words]

Design [Max: 1000 words, currently 1042 words]

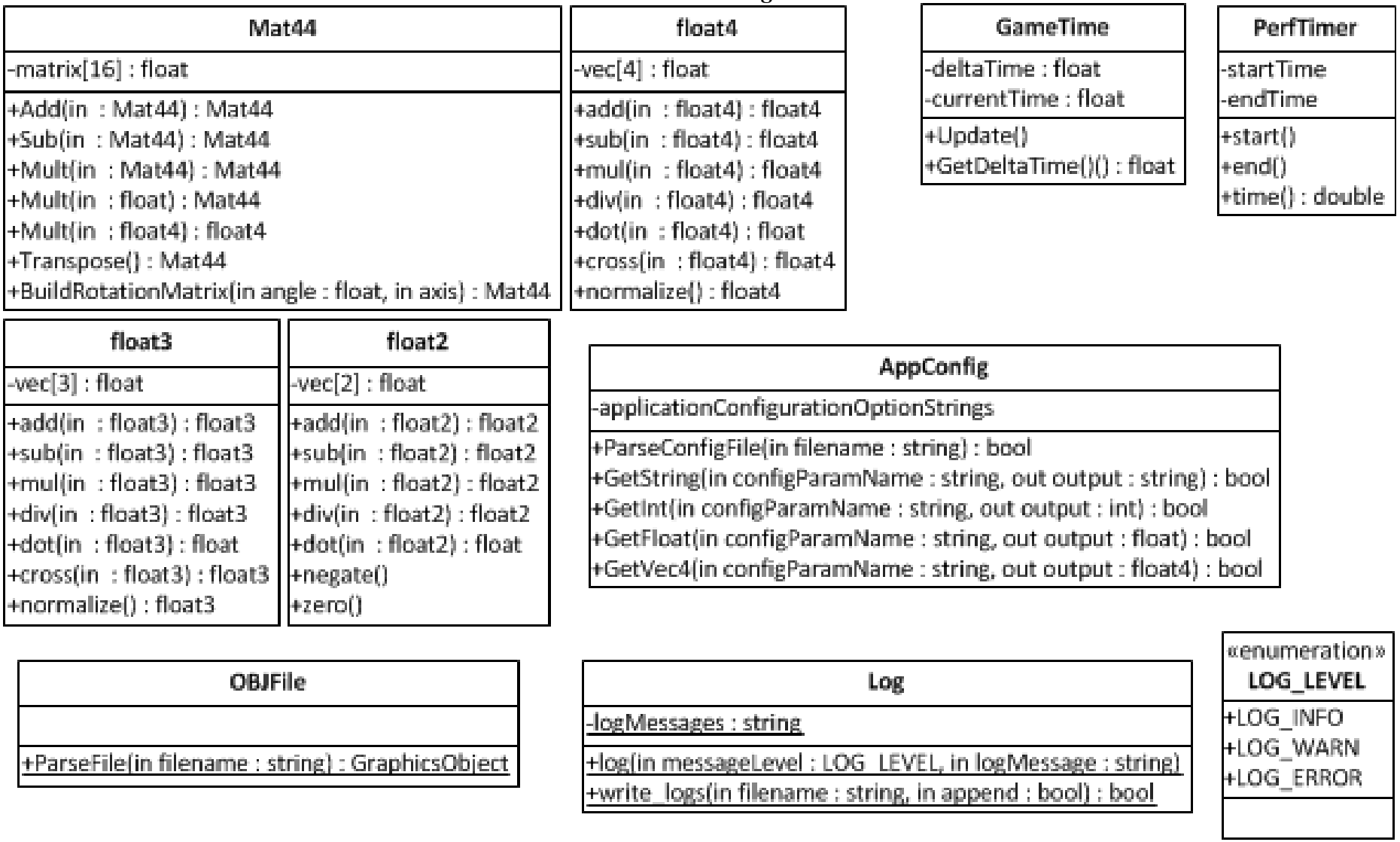
Class and Sequence Diagrams

The following section provides class diagrams for the main classes (primarily those that have a visual influence), utility classes (such as mathematics classes), and sequence diagrams, denoting thflow through the main Draw function, and Load function.

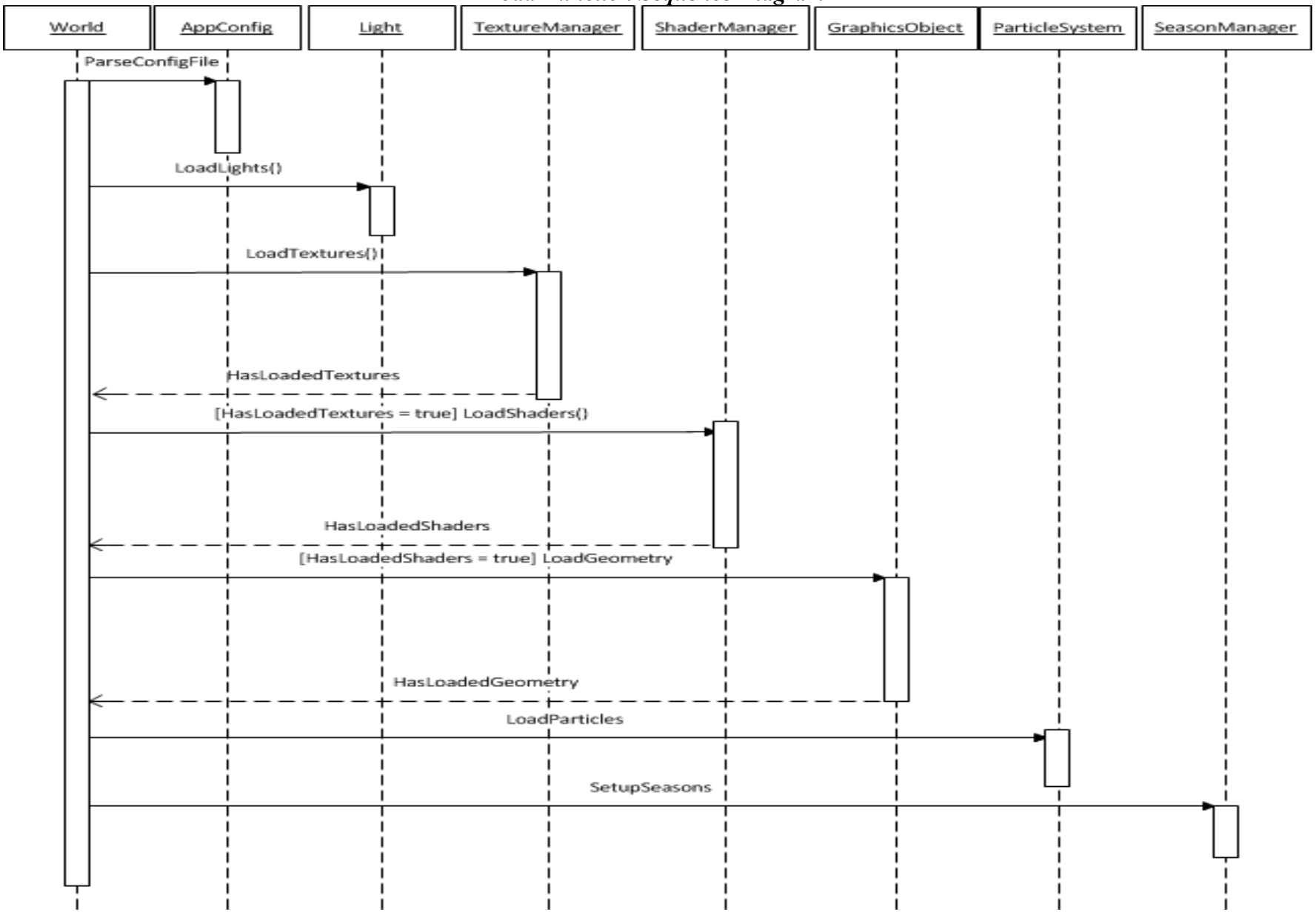
Main Class Diagram



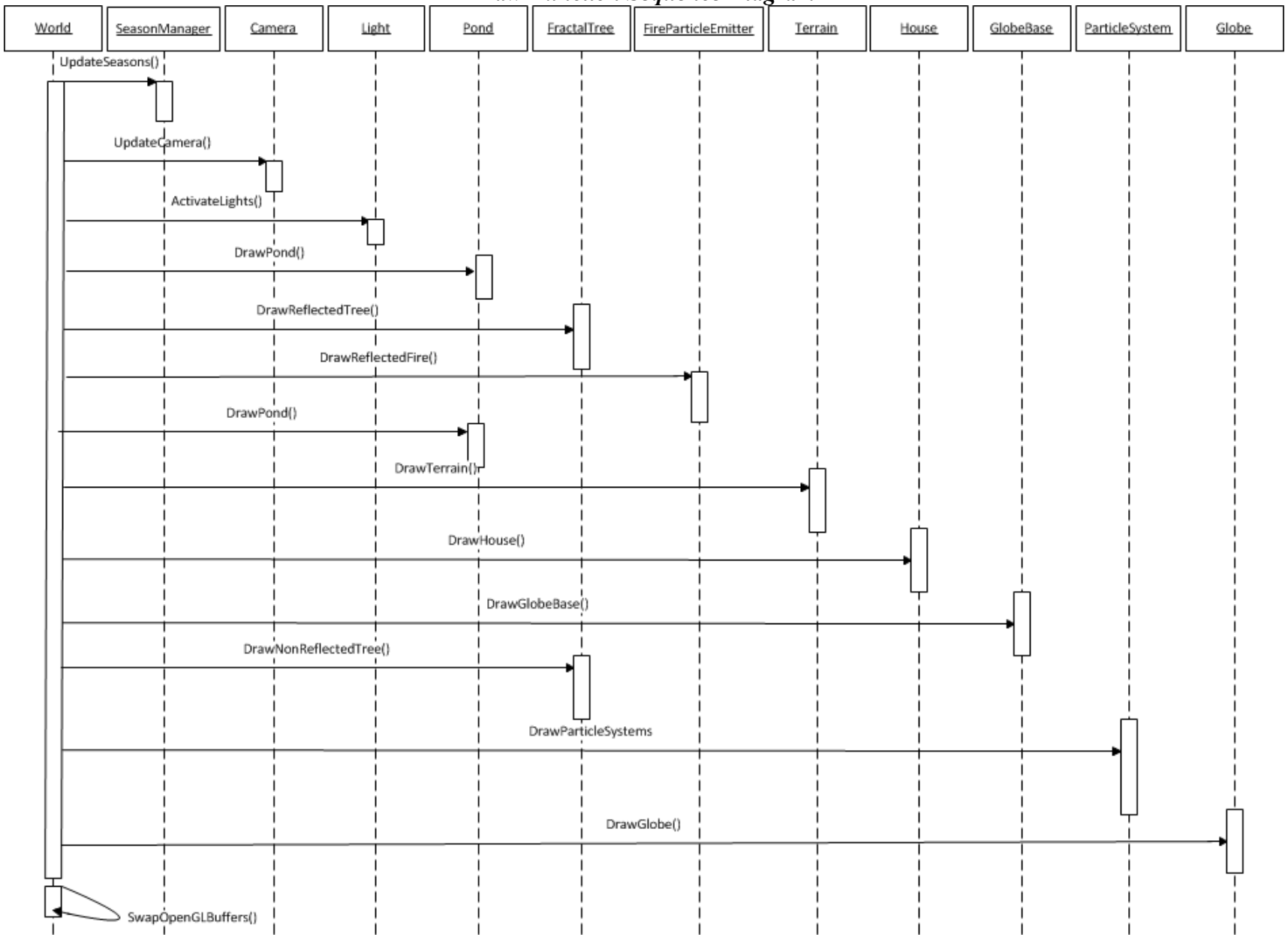
Utilities Class Diagram



Load Function Sequence Diagram



Draw Function Sequence Diagram



The System Classes

This section outlines the roles and responsibilities of each non-trivial class within the system.

Geometry Classes

The **Model** class is the only class to directly store and manage geometric data, and the creation, usage and deletion of associated Vertex Buffer Objects.

The **GraphicsObject** class collects elements required to render an object, including the geometry, transformations (translation, scale, orientation) and shading parameters (material, textures, shader). Activating and deactivating properties, it contains a Draw function.

The **Cylinder** and **Sphere** utility classes extend the GraphicsObject with properties including radius and height, creating geometry mathematically.

The **TerrainLoader** class (extending GraphicsObject) is responsible for loading the terrain from a height-map, with the indices calculated so a disk is formed.

The **OBJFile** class (used by house and globe base) is responsible for loading geometry from an OBJ file, returning a list of GraphicsObjects.

Fractal Tree Classes

The generic **LSystem** class is responsible for generating a string based on a set of production rules, starting string and generation count.
The **FractalTree** class is responsible for animated growth and death, and Lsystem interpretation to build transformation matrices for the tree.

Mathematics Classes

float2, **float3**, and **float4** are responsible for providing system-wide 2D, 3D and 4D vector operations respectively, including: dot and cross product, addition, subtraction, multiplication, and normalisation.
The **Mat44** class, entirely written with SSE instructions, provides 4x4 matrix functionality, including matrix multiplication. The matrix size was chosen as it matches the OpenGL matrix size, and it can contain any required transformation.

Shading Classes

The **Shader** class is responsible for compilations, activations, deactivations and uniform submission, of vertex and fragment shaders, producing a program applied to a model.
The **ShaderManager** maintains all shaders and serves as a single point of access. It is responsible for addition and removal of shaders.
The **Texture** class, storing the OpenGL texture ID manages operations including setting minification and magnification filters, and texture activation/ deactivation.
The **TextureManager** maintains all textures loaded (from files) and serves as a single point of access, to add or remove textures.
The **Light** class, used to activate lights, contains all light properties (regardless of light type) including position, spotlight direction and cut-off angle, and ambient, diffuse and specular colour.
The **Material** class manages object material properties, including ambient, diffuse and specular colours.
Both **Material** and **Light** activate properties through OpenGL, which can be accessed through built-in shader variables.

Particle Classes

The **ParticleEmitter** class, which cannot be directly instantiated, is the base emitter, responsible for the update and drawing of emitters, providing pure virtual “hooks” used by emitters extending the base class. It contains common data, including a particle array, emitter origin, force set and textures. Classes extending ParticleEmitter include: **HemiSphericalParticleEmitter**, **PointBasedParticleEmitter** (emit particles from point in spread), **StaticParticleEmitter**, and **FireParticleEmitter** (defining particles along a line).
With a handle based implementation, the **ParticleSystem** manages the update and drawing of all active particle emitters.

World Classes

The **SeasonManager** manages events and transitions between seasons, normalizing time per season into a 0 to 1 range, and maintaining a list of events, each with an offset into this range.

The **World** scene manager class stores world state, including GraphicsObjects, lights, shader and texture managers. It is responsible for the global update of systems, and scene drawing.

Miscellaneous Classes

The **GameTime** class, used to control animation speed, tracks time in the application, providing delta time since the previous frame.
The **AppConfig** class loads the configuration file, allowing users to query it for values associated with variable names.

System Design Critique

This section provides critical analysis of the software design, outlining potential improvements.

Merits of the Design

The software design clearly distributes responsibilities among well defined singular purpose classes, allowing easier initialisation, usage, and clean-up, and improving stability and portability of code (due to reduced coupling). Reducing bug potential, few global variables exist, creating a more stable, secure product.

The software design promotes data hiding, encapsulation and maintenance, with clear access interfaces for all data.

Due to the deterministic nature of the application, all memory required is allocated at load time, making memory related issues easier to locate.

Weaknesses of the Design

Despite advantages, notable design weaknesses are also apparent. While “manager” classes are reasonable for maintaining a collection of related data, the resource management classes (shader/ texture manager) should have been amalgamated, with a single well- defined interface, reducing code size, development time and bug potential, while improving maintainability and extensibility.

In retrospect the shader system design was flawed, with under-developed relationships between the Vertex and Fragment shaders. The two should be separated, to improve code clarity and promote object reuse.

The application configuration class also had design flaws. Rather than using text parsing and direct string operations, the code could be made more maintainable and extensible by employing classes stream operators.

Design Changes

The renderer design for the fractal tree changed significantly. Originally, the L-System was evaluated at load time and interpreted per-frame (drawing tree). While simple and clear, it proved too expensive, resulting in the more restrictive format of caching matrices at load time.

Due to implementation difficulties, the terrain loading design was also altered. While previously, only vertices and indices required to make a disk were stored from the height-map, it was altered to store all vertices, then calculating a disk of indices, creating a more flexible loader with no missing geometry.

The design of shader uniform set-up was also simplified. Rather than entire manual control, texture, material and light uniforms are set through OpenGL variables (automatically passed to shaders for access).

What I would do differently

If revised, the geometry loading mechanic would be restructured, to serialize geometry data (save in binary engine format) then stream directly into memory (reducing load times and increasing maintainability).

Once OpenGL had initialised, memory allocations slowed significantly. To remedy this, I would add a custom allocator, that allocated a large block of memory, then manually partitioned it for various purposes. Though wasting memory, it would improve load times, and force greater programmer discipline.

To improve system performance, the design of the particle system would be changed to use point sprites, or a custom geometry shader based implementation.

Finally, I would adopt entirely manual control of shader variables and support vertex attributes, remove limitations OpenGL imposes (e.g. 8 lights).

Graphics [Max: 1000 words, currently 996]

Fractal Tree

The fractal tree proved an excellent opportunity to demonstrate problem solving abilities, and an appreciation for algorithmic complexity and performance oriented techniques.

In order to achieve my vision, I wished to be able to easily modify the tree structure, and allow fire to be animated in a semi-realistic fashion, along its branches.

The fractal tree is based on a deterministic Lindenmayer parallel rewriting system, recursively repacing a string using a set of production rules (mapping an “axiom” to a replacement string). Given “AB” and production rules A → AB and B → BA, in the first generation the string would become “ABBA”.
With user defined production rules, generation count, and starting string, we begin by evaluating the L-System, then interpreting it using the rules:

Axiom	Action
F	Move forward

[	Push matrix stack
]	Pop matrix stack
-	Rotate left on X axis
+	Rotate right on X axis
^	Rotate up on Y axis
V	Rotate down on Y axis
<	Rotate left on Z axis
>	Rotate right on Z axis
L	Draw leaf

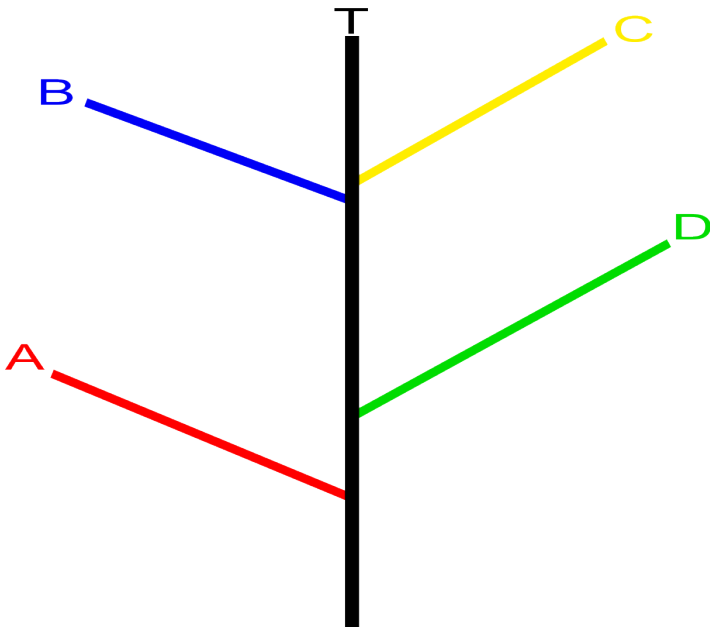
The X, Y and Z axis' are defined as (1,0,0), (0,1,0) and (0,0,1) respectively.

In the original implementation, the L-System was evaluated at load time and interpreted each frame. This was prohibitively expensive and lead to its rejection in favour of caching matrices at load time.

Once the L-System has been evaluated we calculate the tree depth and matrices count for each depth. The depth is incremented and decremented when “[“ and “]” were found respectively. The matrix count is incremented when “F” or “L” is found.

Interpreting each character in the evaluated string, we build transformation matrices and store them in depth-order. Requiring no modifications, these could be used to draw a static tree.

The algorithm required extra complexity to allow animation of the tree/ fire.



Using the original matrix data only, when growing the tree it was found the entirety of A would grow, before B started growing (then C and D). In order to achieve the desired effect (where they would start and finish at the same time), we required information denoting the first and last matrix index per branch segment, for a given depth.

It is then possible to animate growth of the tree over a growth time K. We first calculate which depth of the tree must be animated (scaled). Given the runtime R (growth time so far), and tree depth D, we can find the current animation level:

$$L = \text{lerp}(0, D, (1.0/K) * R)$$

For every depth under L, we draw the branches normally. We must convert the runtime into the range 0 to N, where N is the time per depth (K/D):

$$M = R - (L * N)$$

For each set of branch segments at the current depth, we must find find which [of the multiple] matrix to apply a scaling operation to. Given a segment length S (segment matrix count), the interpolation factor is:

$$F = (1.0 / N) * M$$

The index of the matrix to scale is:

$$I = \text{lerp}(0, S, F)$$

We draw each branch under “I” normally, then scale matrix I (along Y axis) using the fractional part of “I”. All branches above “L” are ignored.

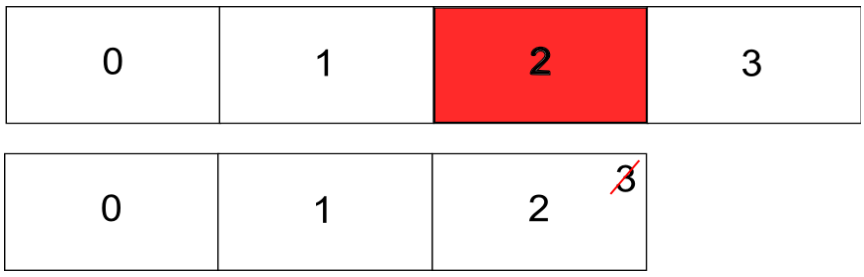
Death of the tree is much simpler – when the fire has engulfed it, we change the alpha value of the appropriate depth. The branches fade out with the fire over time.

Whilst fast and flexible at load time, the implementation of the tree meant it could not be changed once built (without rebuilding the entire tree).

Particle System

The particle system was designed to be flexible in its implementation, while retaining some important performance related characteristics (imposing some limitations upon the user).
The system contains two components: a particle emitter manager and emitters.

The emitter manager is used to manage the storage, update and draw of all emitters in the system. Using a template function (so one may add any of the multiple emitter types), the user may add an emitter. A new emitter is created (on heap), and is added to a list of emitters (a vector of pointers). An emitter handle is also created (in a separate vector), containing the same pointer. When the function completes, the user receives an integer ID (an index into the vector of handles). This ID is then used to retrieve the emitter from the manager. This design was employed, so that an emitter may be removed (with memory deleted), whilst the ID will still remain valid – on removal, the emitter handle pointer is simply nullified. Emitter handles are never removed as this would invalidate emitters passed to the user, if the ID was greater than that of the one removed (illustrated below).



Clearly, if element 2 was removed, the index “3” would become invalid.
The particle emitters themselves inherit from a base Particle Emitter to add specific functionality. The base emitter contains shared data, including: the particle array (static size), billboard type (spherical, cylindrical or none), local particle maximum, origin, rate of emission, forces list, the shader, alpha and colour map. Further, the statically linked functions Draw() and Update() are used to display and update particles. From within the Update function, multiple virtual functions may be called, including an Emit() function (to initialise the properties and emit a particle) and UpdateParticleProperties() function (to update properties that may not be updated in the primary Update() function).
Multiple custom emitters have been defined, including a hemispherical emitter (emitting particles over a hemisphere of a given radius), a point based emitter (emitting from a point in a spread), static emitter (used for leaves), and the fire emitter. The fire emitter uses a list of Particle Lines, where each line has a start and end position, and is associated with a depth in the tree. With this data, it is possible to animate the fire particles on the tree (in a similar fashion to the tree growth function described above).

Project Management [Max: 500 words, currently 476]

The Seasonal Globe project was managed successfully and productively, and by learning the use of new tools and employing existing skills, the final application met both global and personal requirements.

In order to manage time a schedule was created (see Time Plan). This helped motivate and measure progress at any time (comparing current results to expected results). Compared to the archaic methodologies however, I do not believe requirements can be perfectly defined in a single project stage. Instead, I followed agile development methodologies, assessing progress and selecting tasks to complete and review in a short time period. By assignment conclusion, it was clear the schedule had proved useful, and allowed me to improve my scheduling skills and become more disciplined. However, as may be expected, predicted time management is not perfect. Despite best intentions, I found that in the pursuit of a personal vision (and the difficulty of certain implementations), some aspects took longer than expected (primarily animated fire and terrain).

For many years, I have used (and advocated use of) version control systems (VCS). I have appreciated their usage scenarios, and the aid they provide in rapid development and the maintenance of project history and data integrity. At many times I have prototyped ideas – rather than modifying the base code (which may lead to instabilities), VCS' have allowed for cheap “branches” to be created to allow for faster development. Since moving from Subversion (centralised) to Git (distributed), I have also found my work flow to be faster (entire repository copy stored on the client).

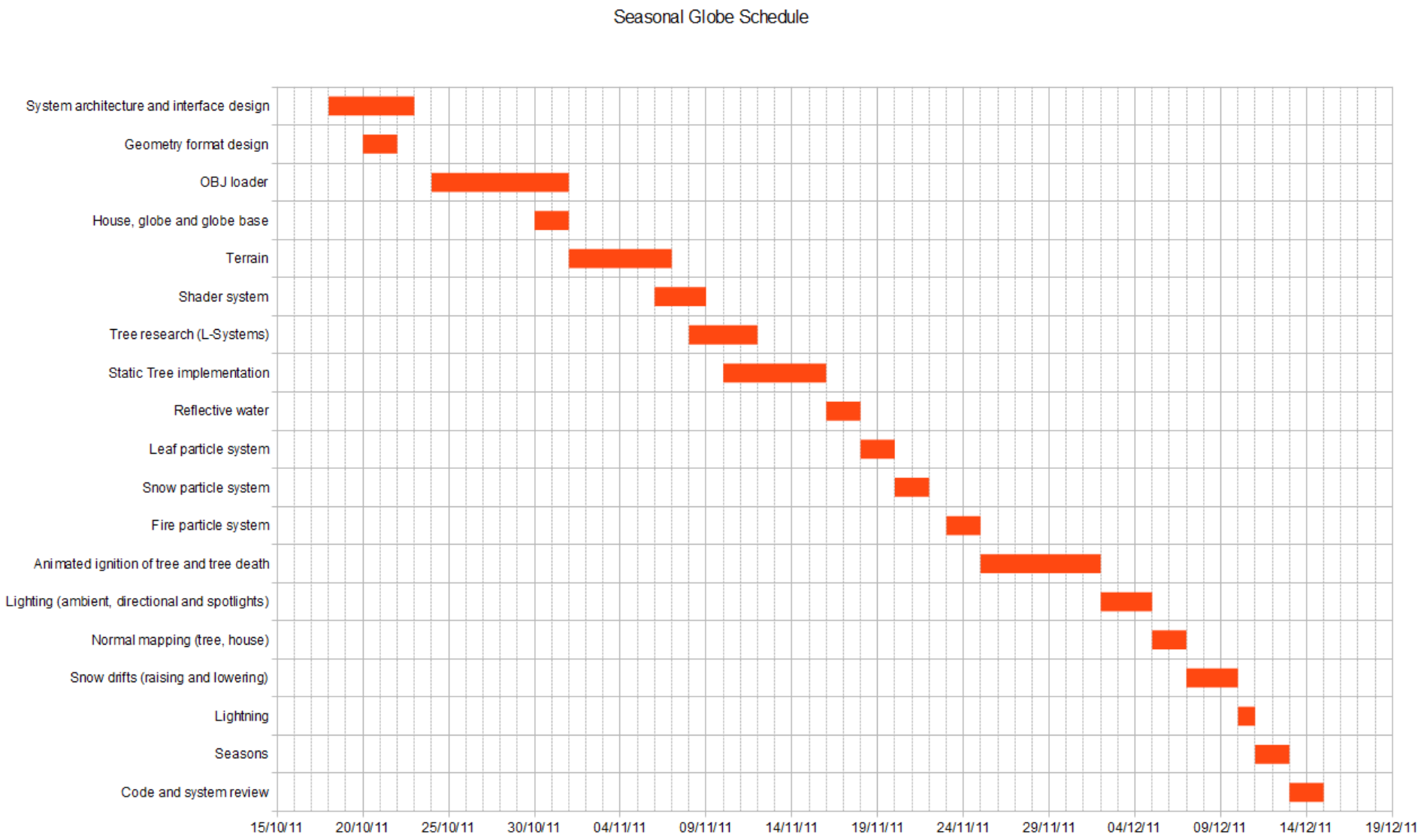
I also employed the project management tool Redmine, which allowed for features such as repository viewing, bug tracking and a “wiki”, entirely accessed through a web browser. As a single centralised source of important information, this lead to productivity gains.

I have also improved my requirements analysis and design skills. Historically, when I have immediately started implementing an application, the lack of design foresight has lead to problems.

Considering this, I chose to split the specification into its component parts. Each area in turn, I designed the relevant algorithms at a high level, rewriting specifications to add further details. Selecting an appropriate algorithm, these were designed on paper, obvious problems being identified quickly and resolved (without the expense of finding issues during implementation stage). Once adequately completed, algorithms could be implemented without many of the concerns that may have arisen without such designs. By following a disciplined approach, my project management and professional skills have improved.

Finally, during the project I discovered the usefulness of static code analysis tools, which I will continue into the future. Using “Parasoft”, I was able to view and resolve design issues my coding style had inflicted. Though some issues seemed of little relevance (in context of use), I believe the static analysis tools overall improved my coding style and the quality of the product.

Time Plan


Seasonal Globe Schedule
System architecture and interface design
Geometry format design
OBJ loader
House, globe and globe base
Terrain
Shader system
Tree research (L-Systems)
Static Tree implementation
Reflective water
Leaf particle system
Snow particle system
Fire particle system
Animated ignition of tree and tree death
Lighting (ambient, directional and spotlights)
Normal mapping (tree, house)
Snow drifts (raising and lowering)
Lightning
Seasons
Code and system review
15/10/11
20/10/11
25/10/11
30/10/11
04/11/11
09/11/11
14/11/11
19/11/11
24/11/11
29/11/11
04/12/11
09/12/11
14/12/11
19/12/11