Seasonal Globe Report

Design [Max: 1000 words, currently 1849]

The System Classes [currently 1047 words]

This section outlines the roles and responsibilities of each important non-trivial class within the system, ordered by purpose.

Geometry Classes

The **Model** class exists simply to store and manage geometry data (triangles) sent to the GPU. Unlike some others, the geometry class performs no data modifications (e.g. it does not apply transformations to the data such as rotations). It is however the only class that maintains geometry data – all other classes that wish to draw an object do so through a Model object. The class is responsible for buffering vertex and index data, activating, drawing and deactivating Vertex Buffer Objects.
The **GraphicsObject** class is responsible for clearly collecting and managing multiple disparate elements that may make up a final displayed object. These properties include the Model object (actual geometry), model transformations (position, scale, rotation) and shading parameters (textures, shader, material). GraphicsObject also contains a Draw function, so that any object inheriting from GraphicsObject may itself be drawn.
The **Cylinder** and **Sphere** classes exist, inheriting from GraphicsObject. The geometry is created mathematically (rather than using a loaded object). The Cylinder and Sphere exist to extend the GraphicsObject with properties including radius and height.
The **TerrainLoader** class (also inheriting from GraphicsObject) is responsible for loading and drawing the terrain. It is loaded from a height-map, with the indices calculated so a disk is formed (suitable to be placed within the snow globe).
The **OBJFile** class contains only static functions, and is responsible for loading the geometry from a WaveFront OBJ file, returning a list of GraphicsObjects. The house and globe base use this class.

Fractal Tree Classes

The **LSystem** class is a generic L-System class and responsible for generating a string based on a set of production rules, starting string and generation count. The LSystem class is responsible for the generation of the string to be evaluated to produce the fractal tree.
The **FractalTree** class is responsible for evaluating the string produced by the L-System, and interpreting each character so that the tree may be formed. It is responsible for the transformations and shading, animated growth and death of the tree.

Mathematics Classes

The classes **float2**, **float3**, and **float4** are responsible for providing system-wide 2D, 3D and 4D vector operations respectively. These operations include dot and cross products, addition, subtraction and multiplication, and vector normalisation.
The **Mat44** class provides 4x4 matrix functionality. The class, primarily used by the Fractal Tree system, entirely uses SSE intrinsics (for performance), and allows for matrix multiplication operations. The 4x4 matrix size was chosen as it can contain any transformation we require (translation, scale and rotation). Further, all OpenGL matrices are 4x4 matrices.

Shading Classes

The **Shader** class is responsible for managing the compilation, activation and deactivation of both the vertex shader and fragment shader, that must be collected to produce a shader program (that can

then be applied to a model). The class is also responsible for the retrieval and submission of uniform values to OpenGL.
The **ShaderManager** manages all shaders in the program, and serves as a single location where all shaders can be found (and cleaned up on application exit). The manager is responsible for the addition and removal of shaders, while maintaining the integrity of existing data and handles passed to the user (shader list index).
The **Texture** class is responsible for the management of single-texture operations. Storing the OpenGL texture ID (rather than any texture pixel data), it is able to set such properties as the Minification and Magnification filters, and activate and deactivate the texture.
The **TextureManager** is responsible for storing all textures loaded (from files) in the program, and texture data clean-up. Serving as a single point of access of textures, it allows for trivial clean-up of data later.
The **Light** class is responsible for the management of all light properties (regardless of light type), and the activation of lights in OpenGL. Properties include position, spotlight direction and cut-off angle, and the ambient, diffuse and specular colour.
The **Material** class is similar to the Light class, and manages the material properties for an object (including ambient, diffuse and specular colours). As with Light, materials are activated through OpenGL, then the built-in uniforms are used within GLSL shaders.

Particle Classes

The **ParticleEmitter** class serves the role of the base emitter. Containing a number of pure virtual functions, it cannot be directly instantiated, however it is responsible for the update and drawing of any emitters that inherit from it. The pure virtual functions (Emit() and UpdateParticleProperties()) serve as hooks, called from the statically linked update and draw function. The ParticleEmitter contains data shared among all emitters, including an array of particle objects (static size), the emitter origin, a set of forces and textures.
Inheriting from ParticleEmitter, other classes provide precise functionality, including: the **HemiSphericalParticleEmitter**, the **PointBasedParticleEmitter** (emit particles from a point in a given spread), **StaticParticleEmitter**, and **FireParticleEmitter** (used to define particles along a line, to set the tree alight).
With a handle based implementation similar to the shader manager, the **ParticleSystem** manages the update and drawing of all active particle emitters system-wide.

World Classes

The **SeasonManager** manages the events and transitions between the seasons, by normalizing the time per season into a 0 to 1 range, and maintaining a list of events (with an “executed” state), each with an offset into this range. The SeasonManager contains separate lists for each season to provide clear partitioning of events and simplify the program.
The **World** class is the central class within the application, and is used to store and manage all world state. It stores all models (GraphicsObjects), the lights and controls the instances of all element managers (ShaderManager, TextureManager). The World class is also responsible for the global update of systems, and the drawing of scene elements (through the GraphicsObjects interface).

Miscellaneous Classes

The **GameTime** class is responsible for tracking time in the application. Providing the delta time since the previous frame, it is commonly used to control the speed of animations.
Finally, the **AppConfig** class is used to load the configuration file, as well as allowing users to query it for specific values (in a number of formats). The class loads the file with a mapping between two strings – this lack of formatting allows any requested formatting to take place on a string.

System Design Critique [currently 802 words]

This section of the report provides a critical analysis of the software design, and outlines possible alterations.

Merits of the Design

I believe the software design clearly distributes the system responsibilities among well defined singular-purpose classes. For example, the shader manager is concerned with the management and lifetime of shader objects only. With the clear separation of elements, it allows for much easier initialisation and clean-up of sub-systems – this not only reduces the bug count, but with little coupling, it improves the portability of the code-base (an important consideration in modern software development).

The design also provides very few global variables – this is a highly advantages design choice, increasing the stability and security of the program, and making bugs much easier to locate and resolve. The software design also promotes data hiding and encapsulation. With clear access interfaces for the majority of data, the bug count is reduced, the the program is more maintainable.

Due to the deterministic nature of the application, all memory is allocated at load time – this allows for fast operations, and by containing memory allocations, it makes any memory leaks easier to locate. However, as noted later, memory allocations proved to be slower than ever expected, leading to possible system additions.

Weaknesses of the Design

Despite having advantages, the application architecture and design also had notable weaknesses.

While “managers” are a reasonable option for maintaining a collection of related data, I believe those related to the loading of data (shader manager and texture manager) could have been combined, with a single simple interface – this would have reduced the code size, development time and number of possible bugs, and would have improved the maintainability and extensibility of the code base (for example, audio file loading could later be added under the same interface).

In retrospect, the design of the shader system was flawed. There was an under-developed relationship between the vertex and fragment shader elements. Rather than being contained under a single object, I believe they should have been separated, making for clearer code and better promoting the reuse of shader objects.

I also believe the application configuration class “AppConfig” had design flaws. Rather than using text parsing and direct string operations, I believe the code could be made more extensible (and easier to maintain) by employing the stream operations that can be added to a class.

Design Changes

Over time, the design of the fractal tree display implementation changed significantly. In the original design, the L-System string was evaluated at load time, then interpreted to draw the tree at runtime. While this was simple and provided clearer code, it proved to be too slow in practice. The design was then changed to the more restrictive format of caching matrices at load time.

Due to the difficulty of its implementation, the design of the terrain data format and loading mechanic was also changed. Originally, the data was loaded from a height-map, and only the vertices and indices required to form a disk were stored. In order to fix bugs and add greater flexibility, the design was changed to create a vertex for every pixel in the height-map (the traditional approach), then generate indices for a disk.

In order to simply the program (while imposing no limitations on the application specification), the design was also changed from entirely manual control of GLSL shader uniforms, to using OpenGL inbuilt variables, for textures, material and light properties. These were then accessed directly through built-in uniform variables.

What I would do differently

If revised, the application geometry loading mechanic would be changed – rather than loading OBJ files and height-map data directly, it would be loaded then written out to a binary format suitable to be loaded directly into the engine. This process is known as "data cooking", and would help simplify the program, while speeding up load times.
Further, as was found during the implementation stage, memory allocations appeared very slow once OpenGL and GXBase had been initialised. In order to effectively combat this, I would propose the addition of a custom memory allocator, where a large block of memory was allocated then manually partitioned for various purposes. While this may waste memory, it would improve load times and force the developer to be more disciplined with memory use.
I would also change the design of the particle system to use point sprites or a custom geometry shader based implementation. While the existing implementation was reasonable, it did not perform especially well with a very high particle count.
Finally, I would adopt entirely manual control of shader variables, and add support for vertex attributes. This would allow for attributes such as vertex tangents to be accessed. Further, by entirely switching to manual uniforms, it would allow for more variables than the OpenGL built-in properties allow.

Graphics [Max: 1000 words, currently 1124]

Fractal Tree [725 words]

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

Working to the Seasonal Globe specification, I also set personal goals defining my own vision for the representation of the globe. I set a requirement to be able to easily modify the shape of the tree, and to allow the fire to track along its branches (producing a semi-realistic illusion of the tree on fire).

The fractal tree is based on a Lindenmayer parallel rewriting system that recursively replaces a string using a set of production rules (mapping from a single character "axiom" to a replacement string). For example, given "AB" and production rules A → AB and B → BA, in the first generation the string would become "ABBA". To simplify the algorithm, a deterministic (not stochastic) L-System was used.

With user defined production rules, generation count, and starting string, we begin by evaluating the L-System, producing the final string to be interpreted using the following 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

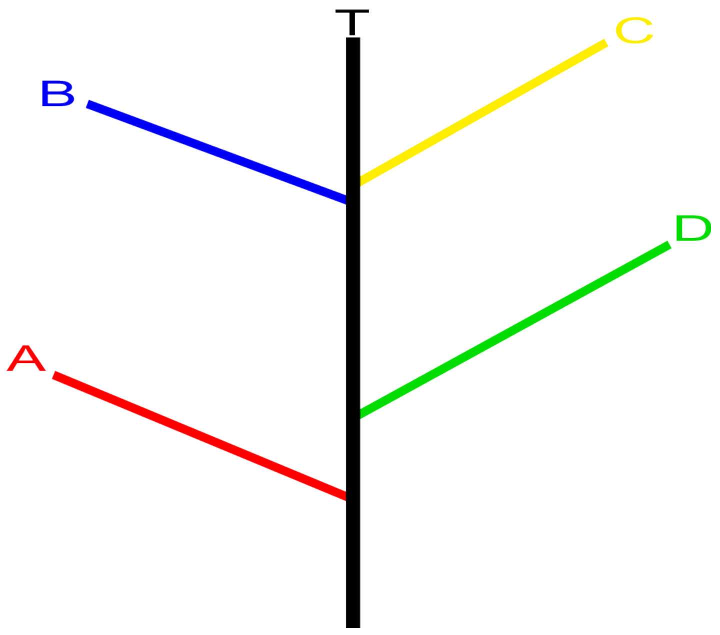
The X, Y and Z axis' are defined as (1,0,0), (0,1,0) and (0,0,1) respectively.

In the original tree 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 pre-built matrices created at load time.

Once the L-System has been evaluated we calculate the tree depth and matrices count for each depth. The depth was incremented and decremented when "[" and "]" were found respectively. The matrix count was incremented when "F" or "L" was found. This allowed us to avoid dynamic memory allocations by defining arrays.

Interpreting each character in the evaluated string, we were able to build transformation matrices and store them in depth-order for later use. Requiring no modifications, they could be loaded into OpenGL before drawing the branch model, producing a static tree.

The algorithm required extra complexity to allow for animation of the tree/ fire. The process and problems of growing the tree can best be described using the following illustration:



Using the original matrix data only, when growing the tree (by scaling the branches at each depth over time), it was found that 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 the growth of the tree over a growth time K. During the animation, we first calculate which depth of the tree must be animated (scaled). Given the runtime R (growth time so far), total growth time K and tree depth D, we can find the current animation level L:

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

For every depth under L, we may 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 exactly which matrix to apply a scaling operation to (multiple matrices make up a segment). Given a segment length S (matrix count in segment), the interpolation factor (used to find the branch to scale) is:

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

The index of the matrix to scale is:

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

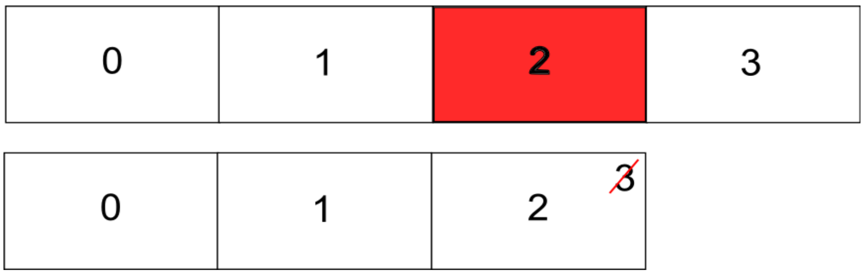
We draw each branch under "I" normally, then scale the matrix at I (along Y axis) using the fractional part of "I". All branches above the level to animate are ignored.
Death of the tree is much simpler – when the fire has engulfed it and started to vanish, we simply change the alpha value of the appropriate 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 [399 words]

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).



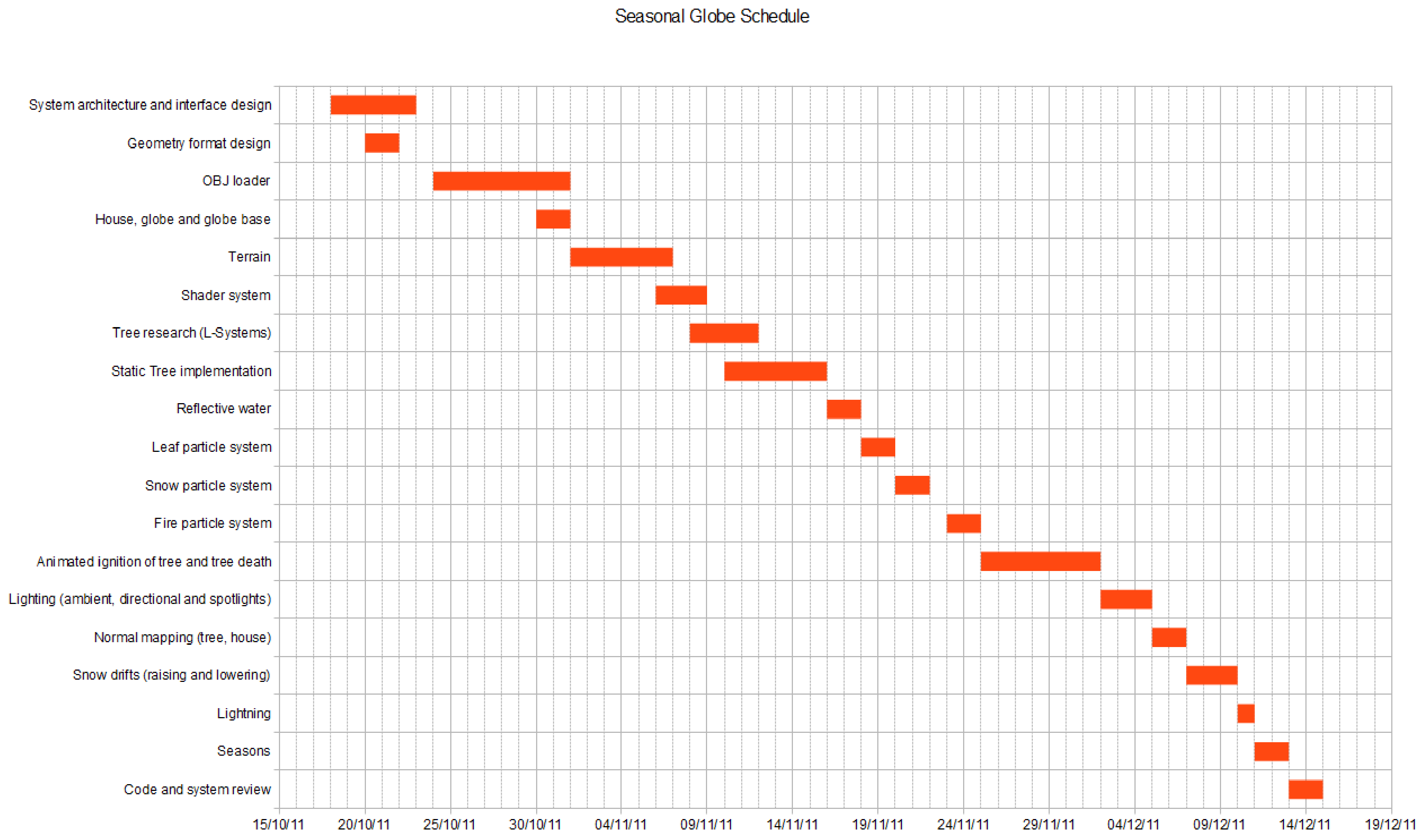
Clearly, if element 2 was removed, the index "3" would become invalid.
The particle emitters themselves inherit from a base Particle Emitter to add specific functionality. The base emitter contains shared data, including: the particle array (static size), billboard type (spherical, cylindrical or none), 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 582]

I believe 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 the global requirements set, and personal requirements.
In order to successfully manage the time of a project, a schedule was created very early on (see Time Plan Gantt chart). This helped provide motivation and allowed me to measure project progress at any time (by comparing current results to expected results). Compared to the archaic Waterfall development methodology however, I do not believe precise requirements can be fully and perfectly defined in a single stage of the project. Instead, I followed agile development methodologies, assessing the progress and selecting tasks to complete in a short space of time. By the conclusion of the assignment, it was clear from the results that the time management 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 areas took longer than expected (e.g. animated fire).
For many years, I have used (and advocated the advantages of) version control systems. With extended use, I have been able to appreciate their use scenarios and the aid they provide in rapid development and the maintenance of project history and data integrity. For example, at many times during the project I have prototyped ideas – rather than modifying the base code (which may in the future cause problems), version control systems 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 (as an entire copy of the repository is stored on the client machine).
As well as Git, 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.
As well as improving time management skills, I have also improved my requirements analysis and design skills. Historically, when I have simply started coding an application, the lack of design foresight has lead to problems in the future. With this in mind, I chose to split the specification into its component parts, allowing for a clear partitioning. Then considering each area in turn, I designed the relevant algorithms at a high level, rewriting specifications to add further details (which may affect implementation). Selecting an appropriate algorithm, these were then entirely designed on paper – this allowed for obvious problems to be identified quickly and resolved (without the expense of finding such issues during the implementation process). Once this had been completed, algorithms could be implemented without many of the concerns that may have arisen without such application designs. By following such a disciplined approach, I believe 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