# Seasonal Globe Report

## *Design [Max: 1000 words]*

## *Graphics [Max: 1000 words, currently 1124]*

### Fractal Tree

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.

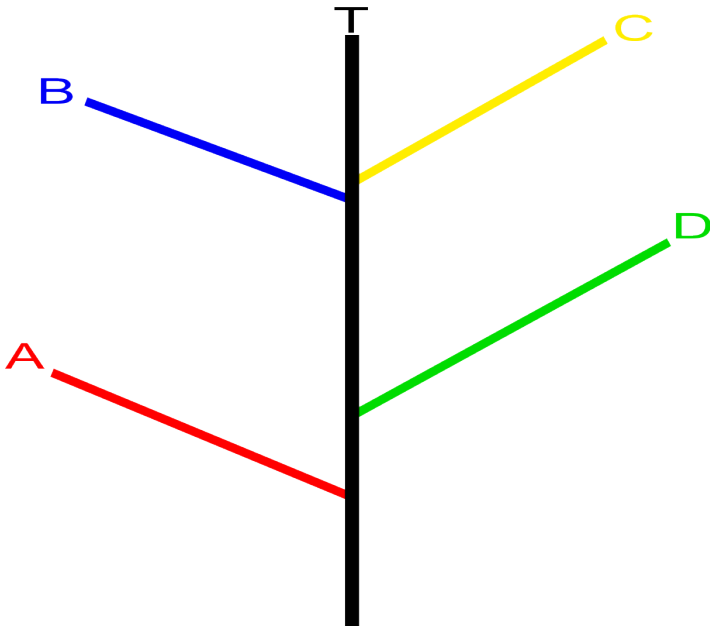
In the original tree implementation, the L-System was evaluated at load time, and interpreted each frame. This was prohibitively expensive and lead to its rejection in favour of pre-built matrices created at load time.

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:

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

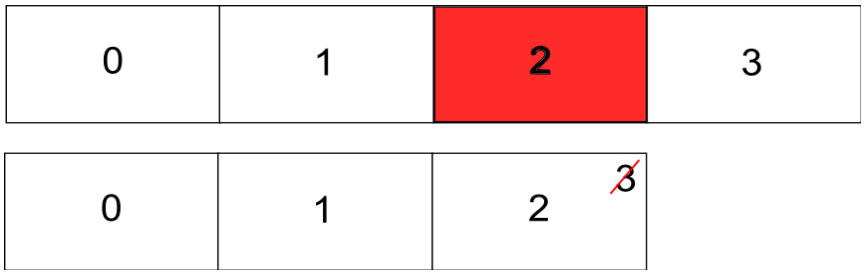
We draw each branch under “I” normally, then scale the matrix at I (along Y axis) using the fractional part of “I”. All branches above the level to animate are ignored.
Death of the tree is much simpler – when the fire has engulfed it and started to vanish, we simply change the alpha value of the appropriate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

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).

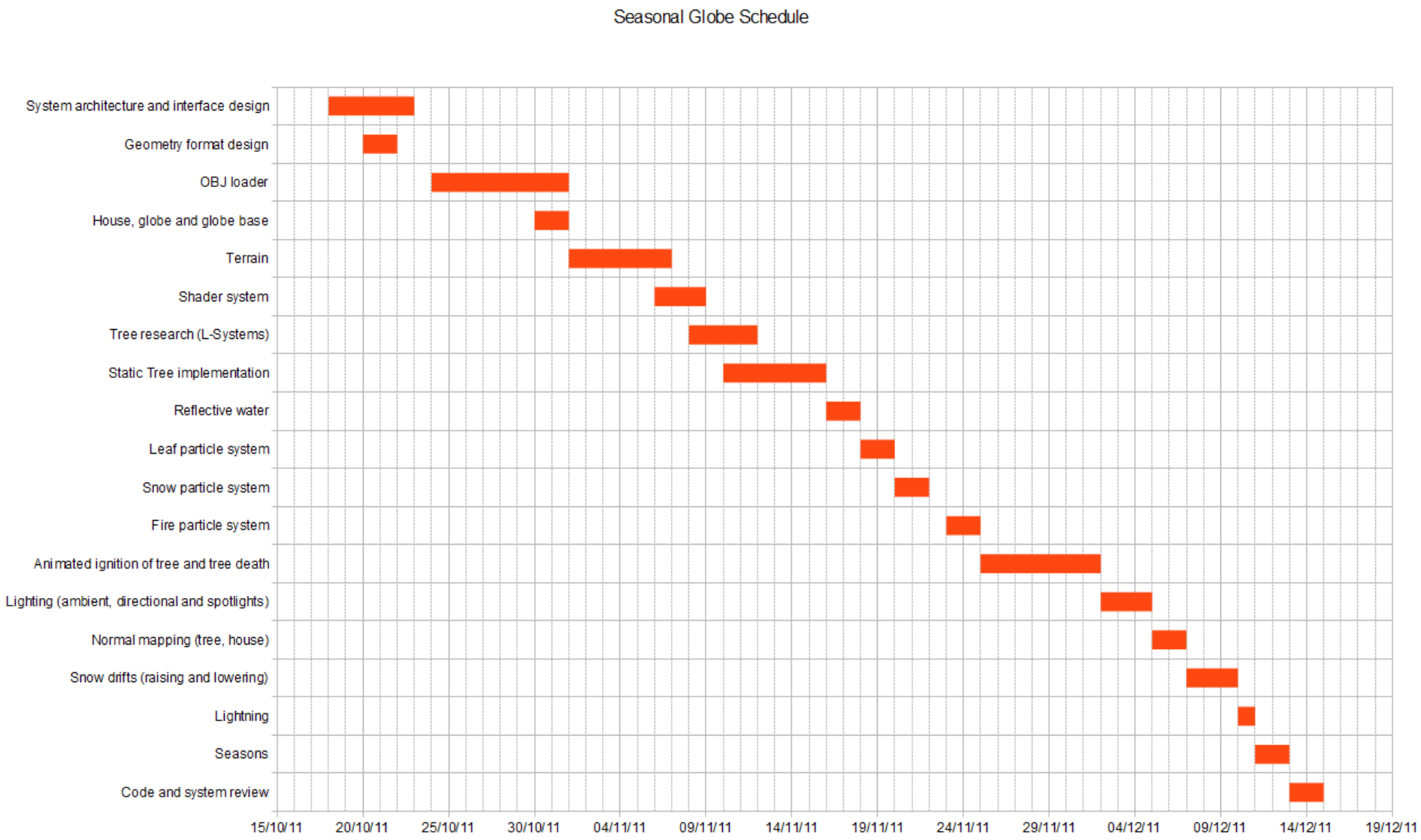
For many years, I have used (and advocated the advantages of) version control systems. With extended use, I have been able to appreciate their use scenarios and the aid they provide in rapid development and the maintenance of project history and data integrity. For example, at many times during the project 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 got to appreciate the usefulness of static code analysis tools, which I will continue to use in the future. Using "Parasoft", I was able to view and resolve design issues that my coding style had inflicted. Although some issues raised seemed of little relevance (in the context of their usage), 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