

# Computer Graphics - Final Submission

Yeun Kim - 12432821  
Šimon Taněv - 12504629

January 2026

## 1 Brief Summary of Implementation

As proposed in our initial plan, we have included a sphere to represent the globe, with a Mars texture applied. A basic Blinn-Phong per fragment shading is implemented in a general shader for object instances. Along with materials and Lights that are sent to the shader along with the mesh, it calculates the shading of objects. The background is a cubemap of a space texture or cloudy day texture. Overall, the implementation features a globe at the center of the scene. A tree is placed on the planet, created using an L-system. From the tree, petals and leaves, generated with a particle system, fall and flow away from the tree to the universe. When the lighting transitions from light to dark, fireworks appear, which are created using a GPU-based particle system.

## 2 Technical Parameters

- Programming Languages: C++ & GLSL
- Graphics APIs: OpenGL for graphics
- Frameworks: Basic framework with a simple camera and shaders
- Libraries: GLFW, GLAD, GLM
- External Assets:
  - Mars texture
  - Cubemap texture (Night)
  - Cubemap texture (Day)
  - Bark texture
- Tested graphics cards:
  - Intel(R) Iris(R) Xe Graphics (on 13th Gen Intel Core i5-1335U)
  - NVIDIA GeForce RTX 4060
  - NVIDIA GeForce GTX 1060 (Vislab PC)

### 3 Controls

When launched, the Demo camera and effects will automatically play (do not exit demo mode for the intended viewing experience). You can switch between democam and freecam using F1. In freecam, you can move using WASD keys, L-Ctrl for down, Space for up motion and mouse movements for camera rotation. Demo does not reset automatically at the end, relaunch the app to the all of the effects again. There may be a camera bug (which was hopefully fixed, but we cannot be sure), which locks the side-to-side movement. Restart the app if that happens.

Some app parameters can be changed using the config.txt file (use defaults for intended viewing experience, # on a newline starts a comment):

1. width - rendered width
2. height - rendered height
3. nearZ - near Z-plane
4. farZ - far Z-plane
5. FOV - field of view
6. leafSwitch - time in seconds when petals switch to leaves
7. treeConfigPath - path to file that describes the tree generation (see L-Systems for more)

The loading of the config file (and tree config file) is performed by our own custom basic parser. The initial plan was to use a library, but that was not used in the end.

## 4 Full Feature Description

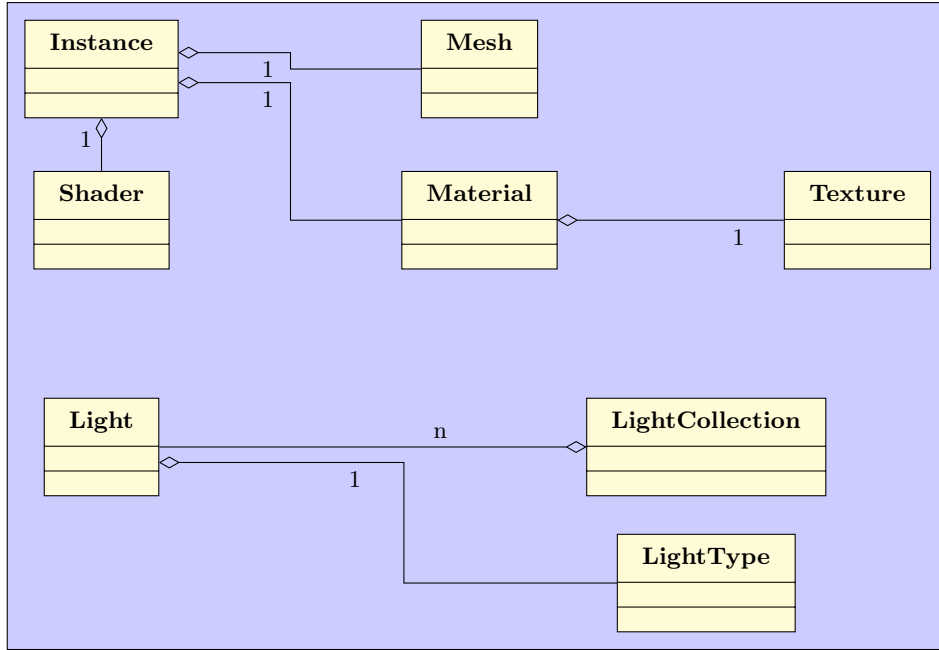
### 4.1 Scene

The scene contains a planet and a tree rendered as a shaded object illuminated by two lights: a directional light from above and a point light, which turns on with the fireworks to create the illusion of fireworks illuminating the scene. The background consists of a skybox with a custom shader that maps a cubemap onto the screen. Petals and leaves are falling from the tree and flowing through the sky. Fireworks appear in the sky as well.

### 4.2 Lighting Model

The lighting model relies on a general object instance, lights, and a shader to do its job; to shade an object using the lights computed in the shader.

#### 4.2.1 General Object Instance and Light System Class Hierarchy



### 4.3 Rendering Pipeline

The draw method of class Instance is responsible for the rendering of the object. It takes in the view matrix, projection matrix and light collection as arguments. It then activates the shader, sends all necessary data to it, and performs a draw call.

The tree is of a child class that inherits the Instance class as it needs to perform the draw call differently. The tree holds two meshes, and two materials (trunk segments and leaves) and a variation of the general shader, which performs instanced rendering, for a single draw call for all of the trunk segments and leaves.

### 4.4 Mesh generation and loading

The sphere mesh is generated by a custom generation code, that builds a sphere, including its normals and uv coordinates. The trunk segment and leaves on the tree are loaded by a custom-written loader from an .obj file.

## 4.5 Skybox

The skybox has its own shader that maps a specially created OpenGL cubemap texture on two triangles that span the entire screen. It includes two textures that go from one to the other in the demo.

## 4.6 Camera

The Camera class provides a first-person navigation system. It allows users to manually move and look around the scene using keyboard and mouse inputs. Manual camera implementation has key features that include:

- keyboard movement in six directions (forward, backward, left, right, up, and down) with calculated delta time and given movespeed.
- Mouse control for smooth yaw and pitch rotation, including clamped pitch to prevent gimbal lock.
- View matrix is generated using `glm::lookAt` to correctly render the scene from the camera's position.
- Perspective projection using `glm::perspective` to adapt the window size for accurate aspect ratio.
- Update function to ensure accurate and responsive orientation changes in a real-time scene.
- Terminate demo mode by pressing F1.
- Automatic movements for demo created by defining control points for a Bézier curve.

# 5 Complex Effects

## 5.1 L-Systems

### 5.1.1 Grammar and String Generation

L-Systems use a formal grammar with a parallel rewriting system to simulate the growth of plants and trees. This implementation uses the basic variation of context free L-Systems to generate the tree in the scene. The implementation is based on the book "The Algorithmic Beauty of Plants" by Aristid Lindenmayer and Przemysław Prusinkiewicz

Grammar symbols can either be a variable, which means that it is only used in the generation of the string, or they can also be a command symbol. Command symbols are also used in the generation of the string, but they also correspond to a command that will be interpreted by the Turtle that traverses the final string to generate the tree itself.

List of implemented commands (with some inclusions not in the book):

1. F - go forward and build a trunk segment
2. L - go forward and build a leaf (not in original book)
3. f - go forward without building
4. + - turn left
5. - - turn right
6. [ - push current turtle state to stack
7. ] - pop current turtle state from stack
8. ^ - pitch up
9. & - pitch down
10. \ - roll left
11. / - roll right
12. | - turn around
13. ! - decrement diameter of segment
14. ' - increment diameter of segment (not in original book)

### 5.1.2 Turtle and Tree Generation

After the string has been generated, it is passed to the Turtle to be interpreted. The Turtle simply reads the string one symbol at a time and performs the corresponding commands. Any encountered variable symbols are simply ignored. The turtle starts in the local (0,0,0) position, pointing up (0,1,0). Every generated segment or leaf has its model matrix computed at this time, so that it is then only passed straight to the shader, and not computed every frame.

There are two extra rules that are not in the original book, which are used to create a better looking tree. Leaf spawn level is a rule that caps the leaf spawning until the turtle is further up the tree. This is so that leaves do not generate down on the main body of the trunk and is mainly there for artistic reasons. Trunk max spawn level caps the spawning of the trunk segments when the turtle is far up the tree, this is mainly for performance reasons, because the smallest trunk segments are not seen anyway, but take up too much time to draw.

### 5.1.3 Config File and Custom Rules

The grammar rules and settings of the generation are loaded from a tree configuration file, which allows the user to define their own rules and settings.

The string reading is case sensitive and uses the char type to read the string, meaning that symbols that fit into char may be used for the rules (although I would not advise to use white space characters and stick with alphanumeric and some basic special characters, # is used for comments).

List of settings:

1. `scale_x` - x-scale of the starting trunk segment (diameter)
2. `scale_y` - y-scale of the starting trunk segment
3. `scale_z` - z-scale of the starting trunk segment (diameter)
4. `tree_y_pos` - y-position of the tree, if custom rules are used, the user may want to shift the tree up or down
5. `tree_scale` - scale of the whole tree, if custom rules are used, the user may want to change the scale
6. `angle` - angle used by roll, turn and pitch commands, in degrees
7. `scale_factor` - scale factor used by decrement diameter ( $1/\text{scale\_factor}$  is used by increment diameter)
8. `step_size` - how far the turtle will go forward by go forward commands
9. `leaf_spawn_level` - Leaf spawn level cap
10. `trunk_max_spawn_level` - Trunk spawn level cap
11. `startingSymbol` - starting symbol of the grammar
12. `iterations` - number of iterations performed in string generation
13. `production rules` - grammar rules, in the format of: *symbol*  $\rightarrow$  *string\_of\_symbols*

Warning: Only the default rules guarantee the required performance of the demo; tampering with them may result in low FPS. Proceed with caution.

## 5.2 GPU Particle Systems

Originally, the leaves and petals effect was intended to be implemented as a GPU particle system. However, using a very large number of leaves or petals (100,000 particles) would be unrealistic for the scene. Therefore, the fireworks effect was selected instead to demonstrate the GPU particle system implementation.

The fireworks effect is implemented as a GPU particle system where simulation and rendering are executed fully on the GPU.

### 5.2.1 Data Representation and Buffer Setup

Each particle is represented using three vec4 streams:

- Position + remaining lifetime
- Velocity + type, where 1 is rocket and 2 is star
- Color + initial lifetime

To prevent read/write conflicts, the system uses ping-pong Shader Storage Buffer Objects (SSBOs). The compute shader reads from the current buffers and writes updated particles into the next buffers, then swaps indices each frame. An atomic counter buffer is used to compact particles, where every new or alive particle increments the counter and writes itself to the next output slot. The CPU resets the counter at the start of the update pass and reads it back after dispatch to obtain the active count, which is then used as the vertex count for rendering.

### 5.2.2 Simulation - Compute Shader

The total number of simulated invocations is `SpawnRockets + ActiveCount`. Particle behavior is determined by the stored type value (1 = `rocket`, 2 = `star`). When a rocket expires, it is removed and triggers an explosion that emits star particles in uniformly random spherical direction, each with random bright color and speed. Star particles are kept until their lifetime expires. After dispatch, memory barriers are inserted to ensure SSBO and atomic counter writes are visible to the rendering stage, and the atomic counter value is read back to update `activeCount` for the subsequent draw call.

### 5.2.3 Rendering - Vertex/Geometry/Fragment

Particles are rendered in a single draw call: `glDrawArray(GL_POINTS, 0, activeCount)`.

- **Vertex shader:** Reads particle position and remaining lifetime, as well as color and initial lifetime, forwards them to the geometry shader, and computes the `gl_position`.

- **Geometry shader:** Converts particle point into a camera-facing quad by generating four vertices in view space, scaling the billboard based on particle lifetime, and passing color and opacity to the fragment shader).
- **Fragment shader:** Computes the final fragment color and applies pre-multiplied alpha to ensure smooth additive blending.

The effect uses additive blending (GL\_ONE, GL\_ONE) to create bright accumulation typical for fireworks.

### 5.3 Additional effect - leaves/petals

The leave/petal effect is implemented as a CPU-simulated, GPU-instanced particle system. Particle motion and state update are computed on the CPU, while rendering is performed efficiently on the GPU using instanced rendering. Each particle represents a textured quad that is camera-facing and animated independently.

Particle attributes such as position, size, rotation, and fade factor are updated every frame and updated to a GPU instance buffer. A single quad mesh is rendered using `glDrawArraysInstanced`, with per-instance data controlling the transformation and appearance of each leaf.

The vertex shader applies per-particle rotation and size, constructs camera-facing billboards using the view matrix, and passes texture coordinates and fade values to the fragment shader. The fragment shader blends between petal and leaf textures, applies per-fragment lighting, and uses alpha for blending.