Graphics Town Project

Ke Ma

Introduction

The graphics town project was written for project 2 of CS 559 *Computer Graphics*. The core of the project is a program that creates a living town with various things moving around in it. There are some basic requirements the program needs to meet as well as a list of technical challenges we can try. My intention was to create a world that appears very realistic, but can be not necessarily very large and complex. I decided to create a seaside scene without any specific reasons. After working on the project for quite a long time, I found the world to be really a small and simple one, and we may not even call it a town, but it looks nice. See Fig 1.1.

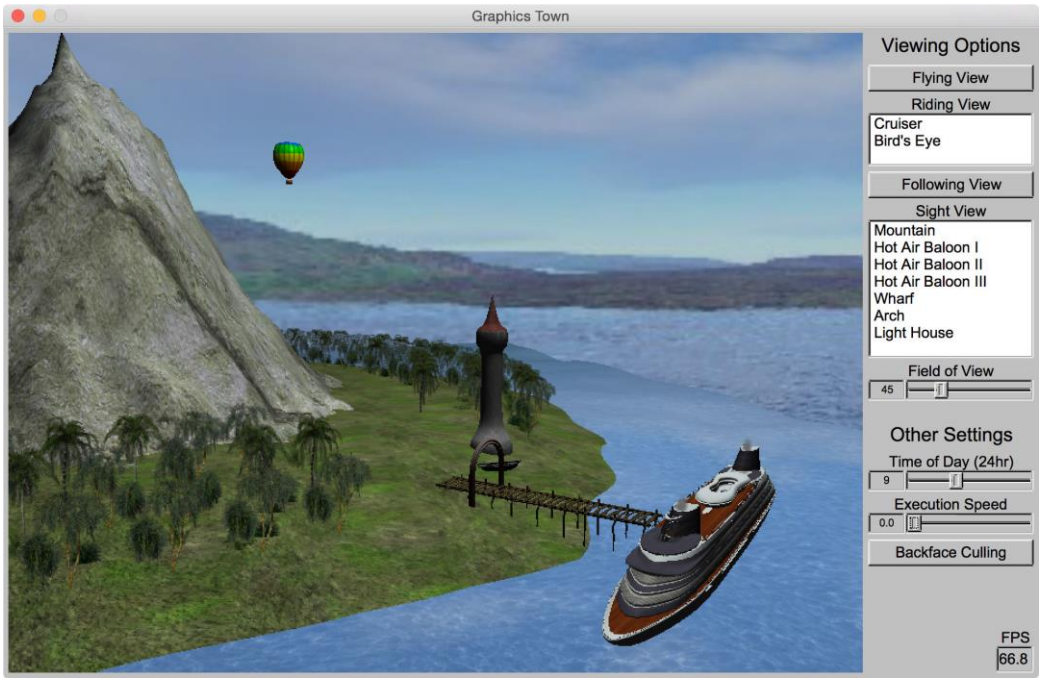

Fig. 1.1.

Basic Requirements

You must have multiple objects moving at any time. There are lots of objects in the world. At least five of them are moving.

You must have multiple types of different behaviors. There are multiple different types of behaviors. The sea is waving. The lighthouse is emitting light in the night. The hot air balloons are floating. The cruiser is sailing and floating. The smoke is coming out of the chimneys.

You must have multiple types of buildings / scenery. There are terrains such as the sea, the island and the mountain. There are plants such as the trees and the bushes. There are facilities such as the lighthouse, the arch and the wharf. There are vehicles such as the hot air balloons, the old boat and the cruiser.

You must have multiple new textures. Among the textures, some must be hand painted, and some must not be flat. There are 28 new textures (including 7 textures used for the models). The textures for the cruiser body and the lighthouse are hand painted (may use other pictures). Almost all of the textures are modified more or less. Almost all of them are non-flat.

You must attempt enough technical challenges. I have attempted more than 10 technical challenges. See the following section for more details.

You must have at least 3 shaders. Among the shaders, at least one must provide a procedural Texture, at least must be affected by the lighting, and at least one must be affected by the time of day. There are 7 pairs of shaders. The one used for the hot air balloons provide a procedural texture (colored horizontal stripes and black vertical stripes). Most of the shaders are affected by the lighting except the ones used for the skybox, the cone light and the particle systems. The ones used for the skybox and the particle systems are affected by the time of day.

You program must work at a sufficient frame rate. The program normally runs at a frame rate of 30~60 if it is built in the release mode.

You must have something that is affected by the time of day. The position and the color of the main light source (sunlight / moonlight) are affected by the time of day. Besides, the lighthouse only emits spot light in the night.

You must use at least one type of advanced texture mapping. I have implemented multi-texturing, environment mapping, bump mapping and skybox. See the following section for more details.

You must have at least an object made out of a curved surface. I have implemented for types of curved surface: surface of revolution (the lighthouse), generalized cylinder (the arch), Loop subdivision (the balloons) and Bezier patches (the sea and the island).

Most of the objects that you made should not use old style OpenGL lighting. None of the objects I made use old style OpenGL lighting.

Technical Challenges

Curved Surfaces

Surfaces of revolution. I implemented the surface of revolution. It takes a planar clamped cubic B-spline curve as the profile curve, and rotates around a fixed axis (normally the Y axis). I used it to create the lighthouse. See Fig. 3.1.

Technical Notes: The clamped cubic B-spline curve is a special cubic B-spline curve that interpolates the first and the last control points. It achieves this by repeating knots at the beginning and the end. The program uses Cox-de Boor recursion formula to calculate the basis functions so as to evaluate the curve. The program takes a set of control points as the input, constructs the knot vector and evaluates the values and derivatives along the curve. When the program needs the value or the derivative at a specific position, it simply looks up the table and linearly interpolates the neighboring values. To create the surface of revolution, the program interpolates along two directions to get vertices, normals and texture coordinates. Some cares must be taken to ensure the normals always point outwards.

Fig. 3.1.

Fig. 3.2.

Generalized cylinders. I implemented the generalized cylinder. It takes a planar clamped cubic B-spline curve as the cross section curve, and a spatial clamped cubic B-spline curve as the axis. I used it to create the arch. See Fig. 3.2.

Technical Notes: To create the generalized cylinder, the program computes coordinate frames along the axis curve. The coordinate frames are constructed with the normal, the bi-normal and the tangent at each point on the axis curve. With these coordinate frames, the programs orients the cross section curve and gets correct vertices and normals. It's a bit tricky to compute consistent normals and bi-normals along the axis curve.

Subdivision surfaces. I implemented the Loop subdivision algorithm. It is able to subdivide any closed triangle mesh. I didn't implement the border cases so it cannot handle open meshes. I used it to create the balloons of the hot air balloons (the baskets are hard-coded meshes). See Fig. 3.3.

Technical Notes: On each iteration, the program first iterates over the input triangles and builds some special data structures. The data structures are maps among vertices, edges and triangles, which makes it easy to look for neighboring vertices. The first step is to move existing vertices by weighted averaging all neighboring vertices that share edges with the current vertex. The second step is to generate new vertices by weighted averaging the vertices of the two neighboring triangles. The final step is to create new triangles by connecting the vertices. The normals and texture coordinates are modified or created in the same way for simplicity, although there may be potential flaws.

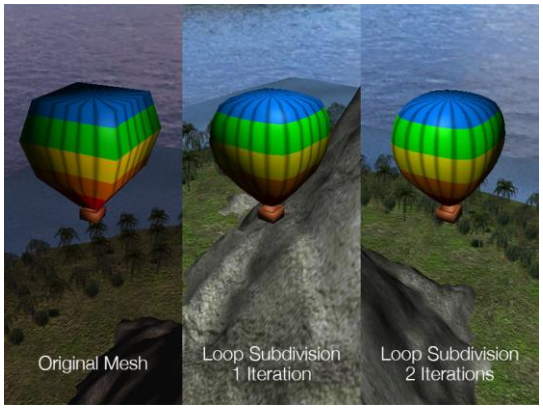

Fig. 3.3.

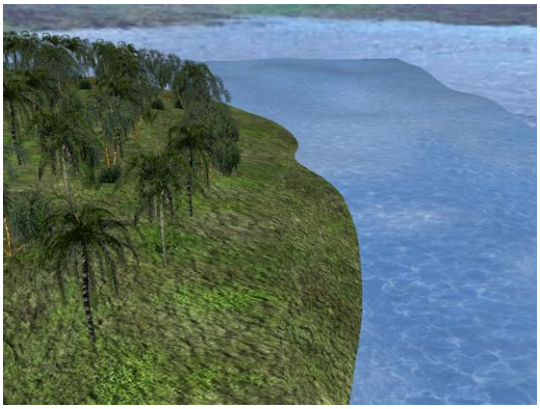

Fig. 3.4.

Parametric patches. I implemented the stitched bi-cubic Bezier patches. The continuity on the edges is guaranteed to be C1. This is achieved by losing degrees of freedom, that is, it uses less free control points. I used it to create the sea and the island. See Fig. 3.4.

Technical Notes: The control points are really carefully selected to ensure the continuity. Suppose there are h * v patches, only 4 * (h + 1) * (v + 1) control points are free. 16 points are related to each patch, but only 8 of them are the control points of the patch. When evaluating each patch, the program first calculates the 16 control points. The evaluation of bi-cubic Bezier patch is equal to evaluating cubic Bezier curve for two times.

Rendering Hacks

Local lights. I implemented local lights that only affect nearby objects. This is achieved with distance attenuation and / or spotlight. I used it to create the light emitted from the lighthouse in the night. See Fig. 3.5.

Technical Notes: The distance attenuation and spotlight equations are implemented in the fragment shaders. They are exactly the same as those used in the fixed-function graphics pipeline. The light emitted from the lighthouse is actually comprised of two light sources. One is a point light with a very large attenuation, which is positioned near the window of the lighthouse. The other is a spotlight with a relatively small cutoff and a moderate attenuation, which only lights up the objects that are in the cone and not very far from it.

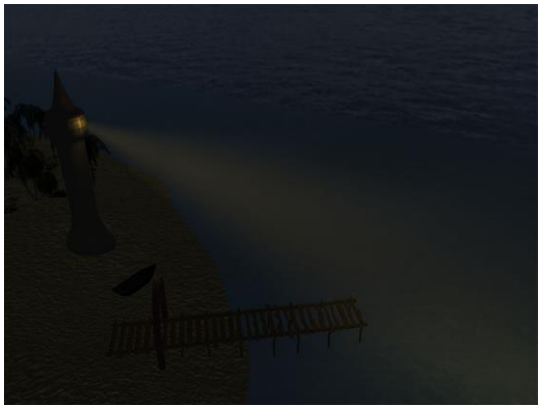

Fig. 3.5.

Fig. 3.6.

Advanced Texturing

Skybox. I implemented the skybox technique. It is always centered at the camera's position and uses a cube map. See Fig. 3.6.

Technical Notes: In order to use the OpenGL cube map, a cube map loader is implemented by modifying the framework texture loader. It loads six images at a time and constructs the cube map. The position of the cube is updated every time the drawing function is called according to the camera's position.

Billboarding. I implemented the axis-aligned billboarding technique. It is a 2D quad that always tries to face the viewer with one axis fixed. It is actually rotating around the axis. I used it to create the trees and the bushes. See Fig. 3.7. Another example of the axis-aligned billboards is the visible light beam. See Fig. 3.5. Another type of the billboarding technique is the point sprite, which rotates around its center. It is used as the particles in the particle systems. See Fig. 3.11.

Technical Notes: The orientation of the billboards is updated every time the drawing function is called according to the camera's position. To keep an axis fixed, the program keeps the axis unchanged in the rotation matrix. The textures used are targa image files with alpha channels.

Fig. 3.7.

Fig. 3.8.

Environment Mapping. I implemented the static environment mapping technique. It uses the same cube map as the skybox. I used it to create the fake reflection of the sea. See Fig. 3.8.

> Technical Notes: The viewing vectors are reflected about the normals to get the vectors that are used for sampling colors in the cube map. It is important to keep all related vectors in the same coordinate system.

Bump Mapping. I implemented the bump mapping technique. It reads normals from a normal map to make the surface bumpy. I used it to create bumpy effect of the island. See Fig. 3.9.

> Technical Notes: Besides the normals, the program also passes the tangents to the shaders. Then the shaders are able to build rotation matrices to transform the normals read from the texture to the normal space to get the new normals. The new normals are used to calculate lighting. Again it's important to keep all related vectors in the same coordinate system.

Fig. 3.9.

Fig. 3.10.

Other Modeling Methods

Fractals. I implemented the diamond-square fractal algorithm. It starts with a regular 2D grid and randomly generates terrain height from the seed values. With different seeds it can generate different types of random terrains. I used it to create the mountain. See Fig. 3.10.

> Technical Notes: The program operates on a 2D array to perform the fractal algorithm. It generates new height values by averaging neighboring heights (in the form of diamond / square) and adding a small random number. The scale of the random number gets smaller after each iteration. To generate a mountain, the program seeds the array using a 2D Gaussian function for three iterations.

Animation (Dynamic Modeling Effects)

Particle Systems. I implemented the particle system technique. It is comprised of a large number of particles that move and fade according to their lives. I used it to create the smoke that comes out of the chimney. See Fig. 3.11.

> Technical Notes: The particles are generated with the same position but with different initial velocities. They move under the effect of their initial velocities and a constant acceleration. Their sizes and transparencies are linearly changed according to their lives. Each particle is a small billboard which is oriented in the vertex shader to reduce computation load. Because the particles involve transparency, they are sorted according to the distances to the camera before drawing.

Fig. 3.11.

Fig. 3.12.

Fake Physics. I implemented some effects of fake physics. The sea is waving. See Fig. 3.12. The cruiser and the hot air balloons are floating.

> Technical Notes: The sea is waving because the heights of its control points are changing according to the time using the trigonometric functions. The floating effect of the cruiser and the hot air balloons is achieved similarly.

Getting Cool Objects

Writing own model loader. I wrote an obj model loader by myself. It is also able to load the mtl material file that is coupled with the obj model file. In this way, one model can have multiple materials. I used it to create the cruiser, the wharf and the old boat. See Fig. 3.13.

Technical Notes: Loading an obj model file is actually a process of parsing a text file and store the data in own data structures. The model loader can handle some cases that are slightly difficult, such as multiple groups / objects, quad faces, relative indices. There are still some deficiencies as well: It cannot deal with faces that are not triangles nor quads; it does not support NURBS surfaces; it does not support some advanced functions like smoothing groups; and so on.

Fig. 3.13.

UI

Better "look at" controls. I improved the "look at" controls. When you're looking at an object, you can adjust your viewing angle using the mouse. The focus and the distance are kept unchanged.

Technical Notes: Just some matrix computations.

Other Things

The cruiser sails around the island and stays at the wharf for a while to pick up passengers.

The light emitted from the lighthouse rotates around it.

There is a bird's eye view with which you can see the whole scene. See Fig. 3.14.

The position and the color of the main light source are carefully designed to be affected by the time of day. See Fig. 3.14.

Fig. 3.14.

Acknowledgements

Some of the resources for this project came from the Internet.

Models

Cruiser is from TF3DM [Link]. It is scaled and simplified before used.

Old Boat is from TurboSquid [Link].

Wharf is from TurboSquid [Link]. It is converted from max file.

Textures

Arch is from [Link]. It is slightly modified before used.

Bushes and **Tree**s are from [Link]. They are slightly modified before used.

Cruiser Body is made by me. The window part uses [Link].

Cruiser Floor is from [Link].

Cruiser Roof comes with the Cruiser model.

Cruiser Tower comes with the Cruiser model.

Cruiser Window is from [Link]. It is slightly modified before used.

Light Beam is from [Link]. It is slightly modified before used.

Island is from [Link].

Island Bump is from [Link].

Lighthouse is made by me. The roof part uses [Link]. The window part uses [Link]. The railing part uses [Link]. The body part uses [Link].

Mountain is from [Link].

Old Boat comes with the Old Boat model.

Sea is from [Link]. It is slightly modified before used.

Skybox is from [Link].

Smoke is from [Link]. It is slightly modified before used.

Wharf comes with the Wharf model.

References

Fundamentals of Computer Graphics, Third Edition.

Real-Time Rendering, Third Edition.

OpenGL Tutorial [Link].

OGLdev [Link].

NeHe Productions [Link].

MTU CS3621 Course Note – B-spline Curves [Link].

Generating Random Fractal Terrain [Link].