CPSC 414, Project 3: YOT - Your Own Thing

Out: Fri 21 Nov 2003
Due: Fri 5 Dec 2003 5pm PST
Value: 15% of final grade

Introduction

Paintball++ was my game: I gave you a specification for the gameplay that you implemented. For this project, you get to create your own game or tutorial, using OpenGL. You can reuse any code that you've already written this term: you've already got the ability to fly, pick, texture, and project. Not to mention the possibly useful ability to model and animate elephants.
The space of possibilities is very broad. Probably any interesting effect that crosses your mind can somehow be incorporated into gameplay!
Specifically, your assignment is to write a 3D video game or tutorial using OpenGL. You are free to design and implement any sort of game you like, as long as it incorporates the required functionality described below. For purposes of this project, we consider a 3D video game to be an interactive 3D computer graphics application that has a challenging goal, is fun to play, and incorporates some concept of scoring or winning and losing. We consider a tutorial to be an interactive 3D computer graphics application that teaches some specific set of concepts. Although originality is welcome, it is not required that your game or tutorial idea be original. For instance, if you want to implement your own (possibly simplified) clone of a 3D videogame you've seen before, that's fine.
The amount of work I expect from you for this project is greater than project 1, and slightly less than project 2. You may form teams of two or three people if you want, instead of working alone. In those cases, the size of the team governs the amount of work required, as described below. Keep in mind you can use the class newsgroup to find prospective team members.

Required Framework

The overall framework of your game or tutorial must include the following features:

- 3D viewing and objects. Your game environment must be a scene consisting primarily of 3D elements, as opposed to only "flat," 2D sprite-based graphics. Your game should provide perspective views of your 3D scene where at least sometimes the viewpoint changes smoothly under some combination of user and program control. To produce these views, you should implement transformation, clipping, and hidden-surface removal of your 3D scene using OpenGL.

- User input. Your game must allow players to interact with the game via keyboard or mouse controls. Alternatively, you can use joysticks or more elaborate user interface devices (provided that you supply the required devices for your program demonstration sessions; see below).

- Lighting and smooth shading. Your game must contain at least some objects that are lit using OpenGL's lighting model. For these objects, you'll need to define normal vectors and materials, as well as create one or more light sources. See Chapter 5 of the OpenGL Programming Guide (the Red Book) for details on implementing lighting and shading.

- Texturing and picking. Your game or tutorial should include at least one textured object and at least one use of picking, in some context that is not identical to what you have done in Paintball++. You already have the implementation of these base functionity, of course. You should use it in some new way in your gameplay or tutorial. You've done textured cubes so far, but that just scratches the surface of how textures can be used for visual richness and to add realism to a scene! Similarly, picking can be used in all kinds of ways to trigger actions other than drawing lines or adding a texture to an object.

New Functionality

The basic requirements above just give you a framework. Again, the specific 7 pieces of functionality that you already have implemented are picking, texturing, lineblasting (scan conversion of lines), flying with the exact camera model of P2, cabinet projection, elephant modeling, and elephant animation.
Your game or tutorial should incorporate (at least) 2 more pieces of new functionality beyond these capabilities. A team of two must do 4 new functions, and a team of three must do 6.
The explicit list below is not at all complete, it's just something to get you started thinking about the possibilities. Considering this list and the amount of effort that it took you to do the seven pieces so far in this class should give you a rough idea of the size of a new functionality piece. You are also allowed to come up with your own ideas for interesting new functionality piece. If you're not sure whether the scope of what you propose is appropriate, ask the professor or the TAs (either in the lab sessions, via email, or via the newsgroup).
Navigation: In Paintball++, you had airplane-style controls where motions were specified with respect to the current camera coordinate system. There are many other possible ways to move the viewpoint. To name a few: orbiting around something

using a "virtual trackball" (p. 203 in the textbook); "terrain following" so that any movement that would go through the floor is ignored, to get an effect similar to walking or driving (see collision detection below); jumping, where you specify an amount of force (say by the size of a mouse drag vector) and then follow some trajectory because of gravity.

Procedural and physically-based modeling: In addition to using scanned or hand-modeled objects to populate the 3D worlds, some video games use procedurally computed models. Chapter 11 of the textbook describes examples of procedural modeling, including fractally-generated mountainous terrains and L-grammars for generating models of plants. Procedurally generated textures may be used to simulate effects such as fire, smoke, and clouds.

Collision detection: Video games often contain moving objects which trigger events when they collide, such as projectiles shot at a moving target. Collision detection can also be used to prevent the user from passing through walls, floors, or other objects. You can implement collision detection in a variety of ways; the simplest might involve comparing bounding volumes of objects to decide if they intersect or not. If you have complicated objects, a hierarchical scene graph (see view frustum culling above) may prove helpful to accelerate your collision detection tests. Chapter 14 of Real-Time Rendering describes a number of collision detection techniques. One particular flavor of collision detection is terrain following, where you ensure that

Simulated dynamics: Your video game implementation might include modeling dynamic behaviors and physics for objects in the 3D world. For example, the wheels of a vehicle might react realistically as they move over rough terrain, or a ball might bounce differently depending on its velocity, elasticity, and the characteristics of the surfaces it hits. Realistically animated characters in 3D graphics applications are sometimes controlled via simulated dynamics.

Particle Systems: Particle systems are simple dynamical systems consisting of many simulated particles which are rendered as points onto the screen. Examples of how you might use a particle system include (1) a fireworks display; (2) a simulated waterfall; (3) a simulated tornado; (4) explosions: if your game involves any sort of mayhem, you might want to show objects bursting apart; (5) water sloshing out of a teapot spout (the teapot itself is easy, using the glutSolidTeapot primitive). The equations of motion which can be used at each time step to update the position and velocity of a particle are as follows:

P = P + V*dt + 0.5*A*dt*dt V = V + A*dt

where P, V, and A are the current position, velocity, and acceleration of the particle, expressed in 3D coordinates. The acceleration, A, is given by gravitational acceleration in the absence of other forces. If you are doing the tornado simulation, you may simply want to directly define the particle trajectories instead. Use GL_POINTS to render the particles, and use glPointSize() to control the size of the rendered particles. For additional sophistication, have your particles can bounce off surfaces or simply run along surfaces, such as water particles before and after the waterfall. In all cases you will want to add some randomness to the initial positions and velocities of the particles in order to get a realistic distribution for the motions of a large number of particles. You should construct your simulation so that you can specify a maximum number of particles that can exist at any point in time. A common way of enforcing this restriction is to give particles a limited 'lifespan', and at steady state you create one new particle for each old particle that you remove from the simulation. You could add various modes that the user can select with keystrokes, which can be used to interactively change various properties of the particles (colour, speed, etc.). See Chapter 11 in the textbook.

Level of detail control: One way to limit the number of 3D primitives drawn each frame is to implement level of detail (LOD) control in your game. One simple method of LOD control involves creating multiple versions of some of your 3D objects, varying in geometric complexity (such as 10, 100, and 1000 polygons). Then, before drawing the objects each frame, you can pick the most appropriate version of the object to render, depending on such metrics as the distance of the object from the viewer, the complexity of the current scene, or a user-selectable detail level. Levels of detail are discussed in section 9.8 of the Real-Time Rendering book.

Advanced rendering effects: OpenGL makes it possible to easily implement a wide variety of realistic rendering effects. Some of these effects can be achieved by drawing the scene multiple times for each frame and varying one or more parameters each pass through the scene; these techniques are called "multi-pass rendering". Other techniques combine traditional 3D graphics rendering with 2D image-based graphics. Advanced rendering effects you can add to your video game include soft shadows, reflections, motion blur, depth of field, bump mapping (if you didn't already get credit for this in project 2), environment mapping, billboarding, and projective texturing. See chapter 6 and 8 of Real-Time Rendering as well as the pointers on the assignment web page for examples and information on implementing advanced multi-pass and image-based rendering techniques.

Sound: Adding appropriate audio effects to your game can provide a more compelling experience for the player. The details of sound effect creation and implementation of audio playback in your game engine will depend on your hardware configuration; some pointers to relevant documentation and sample sound effects are included on the assignment web page.

On-screen control panel: Many 3D video games reserve part of the display area for an on-screen control panel, which may include text or 2D graphical elements for user controls, scoreboards, etc. Flight simulator games often superimpose 2D graphical overlays on the 3D world, thereby creating a "Heads Up Display (HUD)." You can implement a control panel for your game using a variety of techniques in OpenGL, such as orthographic projection and the stencil buffer. Text primitives are not explicitly supported with OpenGL, but GLUT and the window system extensions (GLX, WGL, etc.) both provide commands to help render text.

Motion capture animation: Motion capture data is used with many modern 3D video games to allow characters to move realistically. Pointers to some motion capture data files and implementation ideas are on the resource web page below.

Hints and Resources

This project is purposefully designed to be more open-ended than the first two assignments of the course. You will be expected to do a considerable amount of learning on your own, particularly in gaining experience with programming in OpenGL. If there are computer graphics techniques not covered in the course that you would like to implement, we will usually be able to point you to an appropriate source of information. In order to assist and inspire you, there is an extensive set of pointers to resources for creating videogames at this Stanford CS248 page:
http://graphics.stanford.edu/courses/cs248-02/proj3
I point out that the Stanford games are at least four time more ambitious than what's required here, since they have double the time you do and it's worth twice as much of their total grade! So don't panic when you browse through their Hall of Fame equivalent.
This page includes pointers to OpenGL tutorials, information on game development, useful utility programs, and sources for 3D models and other game content. Of course, you are responsible for understanding and implementing your own video game code; sharing code or libraries between teams is not permitted. Using source code, libraries, or executables you find on the Internet (or elsewhere) is permitted only for the limited cases of programming tools and low-level utilities. In particular, borrowing the code that implements basic or advanced game features (as enumerated above) is not allowed. Looking on the Internet for ideas is permitted and encouraged. Even looking at sample code of game features that you find there is permitted, but simply copying that code is not. We expect you to cite all sources of inspiration (Internet or book or human) in your writeup.
The book Real-Time Rendering, by Tomas Moller and Eric Haines, is on reserve at the main library (but under course EECE 478, not CPSC 414).
Interactive 3D graphics programs such as video games place special demands on computer hardware. If your 3D world is particularly large or complex, or if you use certain OpenGL rendering features (such as texture mapping), you will probably need special graphics hardware in order to get real-time performance. Although the CICSR 011 Linux boxes do implement texture mapping in graphics hardware, other platforms (such as your home machine) may not. If you develop your game on a less powerful platform, it will probably be useful to add options to optionally disable expensive features (such as texturing). The performance of your video game is important, so do not implement too many expensive rendering features if the gameplay is negatively impacted! There are some pointers on the assignment web page to sources of information on maximizing OpenGL performance.
Most successful video games include richly detailed 3D models, textures, sounds, and other content for representing the game world and characters. You have several options available in creating the content for your video game. Simple models can be sketched on graph paper, and the coordinates manually typed into your source code. Models can be procedurally generated, as mentioned above. You can use a 3D modeling package and export the model in a format your program can read. The assignment web page will contain pointers to several freely available modeling packages. The GLUT library provides functions for drawing simple 3D shapes (sphere, cube, etc.) Finally, you can find a wide variety of 3D models, textures, and sounds on the web; see the pointers on the assignment web page. These may need to be converted to a format your program can use; there are many free converters. You are allowed to use code from the Internet for loading models.
OpenGL is window system independent, so an additional toolkit is necessary to provide the necessary windowing and input event management; for this we recommend using the GLUT package that you have been using for the other two projects. However, GLUT offers a very limited set of user interface widgets (little more than menus). If you require slightly more extensive user interface functionality, consider trying the MUI, PUI, or GLUI toolkits; they are easy to use, and are written entirely on top of GLUT and OpenGL. Documentation is available on the web (see the resources web page for pointers). You are free to use other user interface toolkits of your choice. However, be aware that the only toolkit that we (professor and TAs) can guarantee to answer questions about is GLUT.
You are also free to develop your video game on any computer platform that supports OpenGL, provided that you will be able to demonstrate your program during the face-to-face grading sessions in the CICSR basement. You can thus either make sure it runs on the Linux boxes in 011, or bring in your own demo machine (which must be all set up and ready to go when your slot starts), or make sure that it runs on some other machine in the basement labs to which you have guaranteed access. In the course of designing and implementing your video game, keep in mind that CPSC 414 is a computer graphics course. Focus your efforts on the computer graphics techniques underlying the game; don't spend the majority of your time on game design or object modeling if the graphics engine will suffer as a result! Finally, don't make your 3D world or game engine so complicated that interactivity suffers. Your game must be playable!

Grading

The new functionality part of your grade will be done with a bucket system: minus, check-minus, check, check-plus, plus. Your grade on this scale will depend on the difficulty of what was implemented. Stated simply, grades will be proportional to effort. To get an A on this assignment, we need to see outstanding effort on at least one advanced feature. In general, although implementing more features will increase your grade, we prefer that you implement fewer features but do them well, and we will grade accordingly.
The rest of your grade will also be done on a bucket system, evaluating success of the project as a whole. For games, this criterion will include playability and visual richness of the world. For tutorials, it includes effectiveness at illustrating and teaching the target concepts.

Writeup and Handin

You need to create a fairly thorough writeup for this project. (Some of you did quite thorough writeups for the previous projects, but some did not.) You must describe the gameplay or tutorial goal in words at a high level, and also include detailed instructions of the low-level mechanics of how to actually play. As stated above, cite all sources of ideas and especially any source code you looked at for inspiration on the Internet.

Follow the same guidelines for file timestamps and such as with project 2, except use the command 'handin cs414 proj3'. Again, only bring hardcopy of the README file to your demo slot (that is, no need to bring code printouts). We will again be grading through face-to-face demos. At least one person per team needs to be at the demo, but having everybody there is optional.

For multiperson teams only: separately from your joint team submission, and privately, each team member should prepare an individual writeup by 6:00pm on December 5 specifying who worked on which features of your game, and what percentage of the total work each team member did. Your writeup should be a single text file, and include the full names and usernames of your team members. Submit this by running 'handin cs414 proj3.priv'. These submissions will be held in the strictest confidence by the course staff and will not be discussed during the face-to-face demo.

The best work will again be posted on the course web site in the Hall of Fame.

subsection*Acknowledgements This project was heavily inspired by Marc Levoy's Stanford CS248 class final project.