CPSC 314, Project 4: YOG - Your Own Game

Out: Thu 26 May 2005
Proposal Due: Wed 1 Jun 2005 4pm PST
Project Due: Fri 17 Jun 2005 11:59pm PST
Value: 18% of final grade

Introduction

For the previous three projects, I gave you a exact specification to implement. 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 flying and terrains, if you would like to use those in your game. You may also use the object loader from project 0. Plus, of course, you have the possibly useful ability to model and animate giraffes. 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 incorporates some concept of scoring or winning and losing, and is fun to play. 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 roughly double the other projects, as this is worth twice as much and you have more time in which to do it. 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.
- **Interactive:** Your game must allow players to interact with the game via user input, for example 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 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.
- **Picking:** Your game or tutorial must include at least one use of picking. Picking can be used in all kinds of ways to trigger actions, including the obvious one of blowing up the object that is picked. If you actually draw your pick ray, it will look like a laser beam shooting out into the scene. Picking must be done in a 3D environment where the viewpoint can change.
- **Texturing:** Your game or tutorial should include at least one textured object. Textures can be used for visual richness and to add realism to a scene.
- **On-screen control panel**: Your 3D video game should use part of the display area for an on-screen control panel, which may include text or 2D graphical elements for user controls, scoreboards, you-are-here maps, etc. For instance, 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.

Advanced Functionality

In addition to the basic requirements above, your game or tutorial should incorporate (at least) 2 more pieces of advanced functionality beyond these capabilities. A team of two must do 4 advanced 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. You are also allowed to come up with your own ideas for interesting advanced 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 Project 3, 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"; "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 modeling or textures: In addition to using scanned or hand-modeled objects to populate the 3D worlds, some video games use procedurally computed models. 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 or people that hit each other. 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 may prove helpful to accelerate your collision detection tests. A future lecture will be on collision detection. One particular flavor of collision detection is terrain following. (Note that you can count terrain following either as Navigation or Collision detection, but not both.)
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.
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.
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, environment mapping, billboarding, and projective texturing. See the pointers on the assignment web page for examples and information on implementing advanced multi-pass and image-based rendering techniques.

Hints and Resources

This project is designed to be more open-ended than the first three 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/cs248-04/proj3/index.html
I point out that the Stanford games are a lot more ambitious than what's required here, since they have more time you do and it's worth more 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. If you cannot explain an algorithm when asked during the demo, you will not get credit for that feature.
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. 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. However, remember that this is a programming not an art class, so do not sink too much time into modelling!
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. If you choose to use sound in your game, it is unlikely that it will be cross-platform compatible between Windows and Linux. Sound will be counted as part of general gameplay, in terms of grading.
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. So you're free to use Windows or MacOS, as long as you can demo it.
In the course of designing and implementing your video game, keep in mind that CPSC 314 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 AI 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 advanced functionality will count for 50% of your grade. Grading will be done with a bucket system: zero, minus, check-minus, check, check-plus, plus. A tentative mapping of buckets into numbers is zero = 0, minus = 40, check-minus = 60, check = 80, check-plus = 100, plus = 105, but I may change this mapping. Note that a bucket of plus corresponds to extra credit, up to a total of 5% in all.
Your grade on this scale will depend on the difficulty of what was implemented. Stated simply, grades will be proportional to effort and success. If you implement more functionality pieces than the minimum, your grade will increase. However, I will grade such that implementing fewer features well is a better strategy than implementing many features minimally. Any of the advanced features can range from something very simple that gets a low score, up to something complex that gets a high score. For example, very simple collision detection like constraining the viewpoint to stay within some absolute distance from the origin of the world would be a check-minus or a minus. Something more involved like checking for intersection of a single object to other objects in the world, like the viewpoint against the flat walls of a maze, or projectiles colliding with objects in the scene, would be a check. Having multiple objects intersecting against multiple objects would be a check-plus. Data structures to support hierarchical collision detection would be a plus.

The rest of your grade, 50%, will also be scored on the same 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. Of course, missing any of the six required features will lower your grade.

Proposal

Your first milestone is a proposal, due by 4pm on Wednesday, June 1. Your proposal can be brief - 1 page is plenty. Be sure to list all team members. The proposal should clearly state the premise of your game, describe the 3D world you plan to build for it, outline the gameplay, and enumerate the 2*N advanced techniques you plan to implement. If you envision facing any special technical challenges (like predicting collisions between a basketball and a hoop), list them.

You need to include at least one image in your proposal - a mockup of a screenshot of the game in action. A scanned pencil sketch will do, or something you whip up in Photoshop, Illustrator, KidPix, or your favorite computer art program. Carefully label the principal components in your image. Sample labels might be: "waterfall [here] will use real dynamics", or "[these] missiles will emit particle system smoke". The purpose of this requirement is to help me understand your proposal, and to help you think about how hard or easy it might be to implement what you propose.

Include a **working email address that you regularly read** in your proposal, so that I can send feedback if I see potential problems. For instance, I might warn you that you're describing something so ambitious that you're unlikely to finish within the available time, or that your proposal is so simple that it will be hard to get anything beyond a low grade, or that you're not specific enough about what you're doing to let me judge. You do not have to wait for my feedback to get started on your project. Your proposal is not a contract - you may well end up changing things if you run into snags with your implementation. If you're making drastic changes and have any questions about whether you're still doing something reasonable, feel free to send me email. However, you don't have to check with me for every little change.

Use the command 'handin cs314 proj3.prop'. Create a directory which contains the text of your proposal (plain text preferred, but we'll accept PDF) and the image(s). If you prefer to do this on paper because you cannot easily scan in a sketch, then hand it in to the 314 box in the CICSR basement.

Writeup and Handin

You need to create a thorough writeup for this project, much more so than for previous projects. Your writeup should have four parts:

- **what:** a high level description of what you've created, including an explicit list of the advanced functionality items
- **how:** mid-level description of the algorithms and data structures that you've used
- **howto:** detailed instructions of the low-level mechanics of how to actually play (keyboard controls, etc)
- **sources:** sources of inspiration and ideas (especially any source code you looked at for inspiration on the Internet)

Follow the same guidelines for file timestamps and such as with previous projects, except use the command 'handin cs314 proj4'. Again, bring hardcopy of the README file to your demo slot. We will again be grading through face-to-face demos, and the grader will be Dr. Munzner. 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 the due date 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 cs314 proj3.priv'. These submissions will be held in confidence by the instructor and will not be discussed during the face-to-face demo. If I have questions, I will email you privately. In most cases, you will both agree that it's close enough a 50-50 split. However, in case of problems, this mechanism allows you to let me know when the teammates have put in substantially different effort.

The best work will again be posted on the course web site in the Hall of Fame. If your term is too busy to spend vast amounts of time on this, then don't. Implement the required functionality and two advanced techniques, at least one of them well, and you'll get an A. Every year, many students do this successfully. Be inspired by your colleagues who are gunning for the Hall of Fame, but do not be intimidated.

Acknowledgements

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