CPSC 314, Project 4: Your Own Game

Out: Mon 19 May 2007
Proposal Due: Fri 23 Mar 2007 3pm
Project Due: Fri 13 Apr 2007 6pm
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 using OpenGL. You can reuse any code that you've already written this term: you've already got many ways to navigate, if you would like to use those in your game. You may also use the object loader from project 0. Plus, of course, you might choose to somehow include animated armadillos in your game. The space of possibilities is very broad. Probably any interesting graphics effect that crosses your mind can somehow be incorporated into gameplay!

Specifically, your assignment is to write a 3D video game 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, I consider a 3D video game to be an interactive 3D computer graphics application that incorporates some concept of scoring, ending with either winning and losing. Although creativity is welcome, it is not required that your game 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. If the team members have different numbers of late days left, the team will be allowed to use the average across all members.

Required Framework: 80%

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

- **3D objects: 10%** Your game environment must be a scene consisting primarily of 3D elements, as opposed to only "flat", 2D sprite-based graphics. You should position objects within the world using transformations. While this is a programming course, not an artistic 3D modelling course, one way to strengthen your game is to have interesting environment with geometric complexity.
- **3D camera: 10%** Your game should provide perspective views of your 3D scene where the viewpoint is smoothly changable through some combination of user and program control. To produce these views, you should be using the standard clipping and Z-buffer hidden-surface removal capabilities of OpenGL. In project 2 you implemented many possible ways to move the viewpoint, and one of those might be suitable for your desired gameplay. Implementing something new (that makes your game more, not less, playable) is again a way to improve your score.
- **Interactivity: 10%** In addition to camera navigation, your game must allow players to interact with objects in the game via user input, most likely from the keyboard or mouse. (You could joysticks or more elaborate user interface devices if you can bring them in for your program demonstration sessions; see below).
- **Lighting and shading: 10%** Your game must contain at least some objects that are lit using OpenGL's lighting model, so you will need to include light sources in your scene and define object normals and materials. You may want to have a mix of lights in world coordinates, to light your whole scene, and lights in camera coordinates for a headlight-style effect. Remember that you can use the ambient term to make the parts of your scene not directly lit somewhat brighter. You may also want to have a mix of lit and unlit objects in your scene, by selectively enabling and disabling lighting. You should make sure that your final lighting is set up so that the user can easily see what's going on.
- **Picking: 10%** Your game must include at least one use of picking. Picking can be used in all kinds of ways in gameplay to trigger actions based on what object is chosen, including but not limited to shooting. Note that if you could actually draw the ray you use for your picking computations, 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 to get credit: that is, so that the mathematics of finding what's under the cursor in the world is more complicated than just using the 2D display coordinate values.
- **Texturing: 10%** Your game should include textured objects - specifically, at least one object must be textured, but you are encouraged to have more. Textures can be used for visual richness and to add realism to a scene.

- **On-screen control panel: 10%**: Your 3D video game should use part of the display area for an on-screen control panel that's always visible. It 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. Hint: you want to draw it in camera coordinates, not in world coordinates! 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.
- **Gameplay: 10%**: Your game must actually be a game: that is, there must be some concept of scoring, and an end where the user wins or loses. You are not required to have multiple levels, with higher ones being more difficult, but that would be one way to make the gameplay more complex (and thus increase your score). There should be something explicit for the user to do, for example actions to take to make progress towards winning the game. Your game should be playable, in terms of the interactive controls being usable to achieve the desired goal. Aesthetics are hard to quantify, but important in having a game that's engaging. Although it's even harder to quantify, think about how to make your game fun!

Optional Features: 20%

In addition to the requirements above, your game should incorporate (at least) 1 option from the list below. A team of two must implement 2 options, and a team of three must do 4.
The list below is not at all complete, it's just something to get you started thinking about the possibilities. You are also allowed and encouraged to come up with your own ideas for interesting options.
Advanced rendering effects: Advanced rendering effects you can add to your video game include shadows (hard or soft), reflections, motion blur, depth of field, bump mapping, environment mapping, projective texturing, and texture billboarding. OpenGL makes it possible to implement a wide variety of realistic rendering effects. For instance, in texture billboarding, a simple object like a square may have a complex texture like a tree. The object always turns within the scene to face the camera head-on, so that the texture is never seen from the side. Having an alpha (transparency) channel is an important part of making this look more realistic.
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. See the pointers on the assignment web page for examples and information on implementing advanced multi-pass and image-based rendering techniques.
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.
Procedural modeling or textures or motion: Particle systems are just one kind of procedral approach. In addition to using scanned or hand-modeled objects to populate the 3D worlds, some video games use procedurally computed models, especially fractal mountains/terrain or L-system plants/trees. Procedurally generated textures may be used to simulate effects such as fire, smoke, and clouds. Procedural motion can simulate flocking behavior of boids that act like birds and fish.
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. One simple form of collision detection is terrain following, so that any movement that would go through the floor is ignored, to get an effect similar to walking or driving.
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.
Animation: Your game could be made more interesting by animating the objects in the world. You could also program smoothly animated camera trajectories. (Feel free to also reuse the animated armadillo from project 1 in your game, but you can't get credit for this option by just reusing that same animation. Similarly, you can't get double-credit for both the animation and particle system options for the same feature, or for animation and procecural motion for the same thing.)

Grading

The required features are worth 80% of your grade, and the optional feature(s) worth 20%.
Grading will be done with a bucket system for each feature: 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, and how successful your implementation is. Any feature 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.
If you choose to have a team of two or three people, you will need to do proportionally more than a single person. A team of two must implement 2 options, and will be graded one bucket higher on all of the required features. That is, the amount of work that would be a check-plus for an individual will be a check for a team. A team of three will similarly be graded one bucket higher on required features, and must implement 4 options.
Implementing extra options can also raise your total grade, by offsetting lower scores on the required features.

Proposal

Your first milestone is a proposal, due by 3pm on Friday, March 23. 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, explain how you are fulfilling the required features and enumerate the options 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. I am not looking for art talent here, just something that will help me understand what you have in mind. Doing this sketch should also help you think about how hard or easy it might be to implement what you propose. Label the principal components in your image so that I know what you have in mind. Sample labels might be: "waterfall [here] will use real dynamics", or "[these] missiles will emit particle system smoke".
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.
You can either hand in your proposal electronically, or on paper. For electronic handin, use the command 'handin cs314 proj3.prop'. Create a directory which contains the text of your proposal (plain text preferred, but I'll accept PDF) and the image(s). If you prefer to do this proposal on paper because you cannot easily scan in a sketch, then put it in the 314 drop box in the ICICS/CS basement (or hand it to me Friday morning in class).
Proposals are graded in one of three buckets: fine, poor, and zero. A grade of fine does not change your total project grade. A grade of poor is a penalty of 1% of your project grade. A grade of zero is a penalty of 5% of your project grade.

Writeup and Handin

You need to create a thorough writeup for this project, much more so than for previous projects. Your README 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)

You are required to include at least two images of your system in action. You are welcome to provide more, or a movie as in Project 1. Follow the same guidelines for file timestamps and such as with previous projects, except use the command 'handin cs314 p4'. Also, please bring hardcopy of the README file to your demo slot. We will again be grading through face-to-face demos, but the instructor will be grading this one. 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 p4.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 be posted on the course web site in the Hall of Fame.

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/cs248-06/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 README 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 011 lab 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 using a 3D modelling package!

Adding appropriate audio effects to your game can provide a more compelling and engaging 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; if you choose to use sound in your game, it is unlikely that it will be cross-platform compatible between Windows and Linux. In terms of grading, sound will be counted towards general gameplay.

Using source code, libraries, executables, or data you find on the Internet (or elsewhere) is permitted, but is limited to geometric models of characters or props (but not entire environments), textures, sounds, and low-level programming tools (like matrix or vector packages). In particular, borrowing code that implements required or optional game features is not allowed, with the exception of texturing and sound. That is, you can use example code as a template for loading textures and sounds. However, you should manipulate texture coordinates on your own. All this said, looking on the Internet for ideas, even looking at sample code you find there, is permitted and encouraged. However, when you do this, I expect you to cite your sources in your writeup. I also expect that your video game was developed only for this course, and that it was begun no earlier than the start of this academic term.

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 011 basement lab. 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 machine in another basement lab 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-oriented features will suffer as a result! Finally, don't make your 3D world or game engine so complicated that it cannot be rendered on the target machine at interactive frame rates. Your game must be playable!

I recommend that you take some time at the beginning to plan your project. First, make sure your idea is of realistic scope, not so ambitious that it's impossible in the time you have, but not so simple that you would get a very low grade even if you do all that you intended. Think about bang-for-the-buck: remember fundamentally you are graded on success, not effort. When you decide on an architecture for your code, think about breaking the work into independently testable pieces. Also think about how to stage your development so that you have a working demo almost all of the time. That is, get simple things working first, and then integrate more complex features into your working demo one at a time. Program defensively: your grade will be higher if you have simple features actually working than if you have complex features that can't be demonstrated because they don't quite work yet. I also recommend that you use version control. If you're working in a team, then following these software engineering best practices is even more critical. For teams, I very strongly recommend integrating pieces of functionality into a common codebase as you go, leaving integration to the last minute is asking for trouble.

If your term is too busy to spend vast amounts of time on this project, then don't. Plan a project of realistic scope, and deliver on it. Be inspired by your colleagues who are gunning for extra credit and the Hall of Fame, but do not be intimidated.

Acknowledgements

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