Project Ox04: Interaction

Progress update Due Sunday, Oct. 19, 2008 at 11:59 pm
Due Friday, Oct. 24, 2008 at 6:00 pm

This project is structured a little differently than the others in order to make the scope manageable. Please read the instructions carefully before beginning work.

For project 4, you will create an interactive real-time 3D system. The example of such a system that you are likely most familiar with is a video game, so the assignment is focused on that application. Other applications of the same algorithms and data structures include training simulators, architectural walkthroughs, 3D modeling systems, engineering simulators, data exploration systems, 3D chat rooms, and cinematic preview systems.

Most commercial games require centuries of person hours, two to three calendar years, and at least $5M in funding to create. So starting from scratch and building a game like Halo 3 or Civilization IV is not something that could realistically be accomplished in an academic setting. The specification for your game is therefore much more modest.

Even creating a game or other interactive system of the scope of this specification would take about two semesters. So, in order to create an interesting system for a project, this assignment is structured with the following aspects in your favor:

1. You will work in self-selected groups of up to four students
2. You have approximately two weeks to complete this project
3. There is no homework associated with this project
4. G3D provides the critical, yet mundane engineering tasks of loading 3D models for you
5. A large number of pre-created art assets are provided in the G3D data directory, including:
   a. Animated character models in data/quake2/ that can be loaded with MD2Model
   b. 3D levels with pre-computed lighting in data/quake3/ that can be loaded with BSPMap
   c. Sky textures and lighting model in data/sky/ that can be loaded with Sky
   d. A variety of props in data/ifs and data/3ds that can be loaded with ArticulatedModel

6. I have pre-designed large parts of the 3D engine for you, and have implemented several pieces.

7. This document contains detailed information about the project design that you can use to split the workload among team members.

8. This document contains detailed algorithm information to augment your textbook (which contains rendering, but not simulation algorithms). Terms in **bold** in this document refer to words with specific meaning that you may find more about in RTR3 and online.

9. You are free to share code for part B of the assignment between project groups, and are **encouraged** to use resources in the library and on-line for any part of the project.

When choosing your team, you’ll need a mix of skills. This includes someone who is very organized to help manage the process and keep the overview documentation up to date; development skills in each of vector math, software design, and algorithm design; and creative and artistic talent/photoshop skills. We all have some skill in each of these areas (with a lot of overlap in the programming area!), but each of us has different strengths and it is important to cover all areas. Also be sure to choose people who have compatible schedules, since even when working on different parts of the program, you’ll need to be in the same room to make effective progress.

1 Specification

Part A:
Implement a real-time interactive 3D system (“game engine”) with the following features:

1. **Instancing:** ability to share a single model among multiple entities
2. Models that can import
   a. Quake MD2 models
   b. IFS files
   c. 3DS files with hierarchies
3. Static world geometry imported from Quake BSP models
4. **Collision detection** between entities and between entities and the world supporting:
   a. Ray-casts against static geometry
   b. Ray-casts against entity proxy geometry (e.g., bounding sphere)
   c. Sphere movement through the static level geometry
      i. simulating a running character
      ii. with no penetration of the world, and
      iii. the ability to run up and down stairs.
   d. Entity-entity proxy geometry collisions
   e. Sphere-world intersection
5. Callbacks (“events”) on entity for collision with each other and for non-**resting contact** collisions with the world
6. Rendering of static world geometry with precomputed light maps
7. Rendering of drop-shadows from entities on the static world geometry
8. Dynamic creation/insertion and destruction/removal of entities from the world
9. User-input to drive the “player” entity using the following keys:
   a. A – rotate left
   b. D – rotate right
   c. W – move forward
   d. S – move backward
   e. Space - jump
10. A follow-camera on the player’s entity
11. A results page discussing:
   a. Your group name and who is in it
   b. The architecture of your system, using data flow or class diagrams and text that is hyperlinked to the major classes
   c. Images demonstrating your engine operating correctly
   d. Key design decisions you made that impact readability, performance, or functionality
   e. Known bugs, with detailed description of how you attempted to resolve or isolate them
12. Documentation for all classes and methods

Part B:
Create a game. Your game must:

1. Have an objective for the player to achieve, such as reaching a certain location or score, beating a time limit, or defeating an enemy.
2. Be (somewhat) engaging and playable by the average CS student.
3. A standalone documentation page explaining how to play the game (i.e., a user manual). Keep this short—at most five paragraphs and some images.
4. A portion of the results page demonstrating and discussing features of the game. The game classes must be discussed in your architecture overview.

I suggest you discuss your game design vision with me early in the process so that I can offer advice and direct you towards relevant resources. It is perfectly fine if you try for an aggressive project and don’t quite make it—you’re going to be evaluated primarily on part A. Finding out why a certain system is really hard to implement is an important part of learning. Plus, you might want to continue this project for your final project, where you’ll have another four weeks to work on it.

1.1 Restrictions
For part A, the normal restrictions in this course apply. You may use any resource except for code written by students outside your group for the current assignment and G3D’s ray-triangle intersection code (which doesn’t provide enough features for this project anyway). You must cite external resources, and are responsible for their quality (i.e., if you use someone else’s ray-cast code, you still have to document and format it for readability, and test it!)

For part B, there are no restrictions on what resources you may use. You may exchange code with other groups as long as you cite their work where it is used and clearly identify what you used in a section of your main documentation page. You can download code from the internet. You can use any piece of G3D. You can use code and algorithms from books. You can talk to other
professors. You can get your friend in Iowa to write a class for you. You can download models and textures from internet sites. You can link against a 3rd party library. You can extract files from your favorite video game DVD. Etc.

Please note that failure to cite the work of others that you used is an honor code violation. I trust you not to intentionally use material inappropriately but it remains essential that you pay close attention to the source of the code and algorithms in your project to avoid the perception of impropriety through carelessness in citations.

1.2 Evaluation

Your work will be evaluated for the following properties, in order of significance:

1. Satisfaction of the specification
   - 80% for part A
   - 20% for submitting anything reasonable for part B
2. Readability, including documentation
3. Mathematical correctness
4. Presentation of results
5. Design (including performance)
6. Programmatic correctness (compiling, memory management, safety)

In the event that you have known bugs, add a documentation section explaining what they are, how you attempted to debug them, and what you think the problem is. Use images as necessary. In general I will not deduct many points for bugs that you are aware of and investigated.

1.3 Progress Report

You will have three kinds of overview documentation this week: your continually updated progress report, your architecture/results, and your very brief end-user game manual. For the progress report, just show some images that make clear what features are working and highlight any bugs. Add small amounts of text explaining what I’m looking at. This is basically the same as the result images you’ve been doing all along. I’ll show some of these in class the following Monday.

Your progress report will be graded pass/fail, which means that you’re all going to pass. I’m requiring it for three reasons:

1. You need to get in the habit of keeping status updates in your documentation for large teams or long projects.
2. This will add some pressure to start the project on time and work steadily, instead of at the last minute.
3. It will help me quickly see where you are and offer appropriate help or suggestions.

For the final project, you’ll be submitting a running progress report every few days. It should not be burdensome—just snap a screenshot whenever you’ve done something particularly neat or are stuck, and add a sentence explaining what is going on. It helps a lot if you keep it in chronological order from newest to oldest like a blog, so that I can always find the latest features on the top.
2 Suggestions for Part B

The simplest game to make is a Quake-style death-match, where characters simply run around the map and shoot each other. I encourage (but do not require!) you to experiment with more interesting mechanics and goals, such as:

1. **RPG**: the character chooses between upgrades throughout the game that make the play experience unique
2. **Quest**: the player must resolve a series of interactions to complete a mission
3. **Resource Management**: balance a set of interlocking commodities and means of production
4. **Puzzle**: solve a physical puzzle, such as moving the character or other object in the world to a specific location or through a series of locations
5. **Relationship**: Characters have simulated attitudes towards one another that are affected by interactions; usually combined with a quest
6. **Stealth**: visibility and simulated noise are liabilities that a character must minimize
7. **RTS/TBS**: combines RPG and resource management, typically where a host of characters are controlled by a single player.

To implement your game, you will likely want to implement some of the following features. None of these are required. These are just suggestions of topics to further research or that might solve specific problems you’ll encounter.

1. **Particle system**: Rain, smoke, fire, spell effects and other collections of particles are best rendered by a particle system. This is an object that renders many billboards, all with the same texture and each represented internally as a single point. At runtime they are rendered using RenderDevice::QUADs with a single begin-end to avoid state changes. For efficiency, particles generally do not perform collision detection. You may need multiple instances of your particle system class for multiple simultaneous effects.

2. **First-person camera**: Put the camera about at the character’s chest and do not draw the player’s own character. If you’d like to show, e.g., what the character is holding, use a special view model that is either distorted or has its own orthographic projection matrix.

3. **Special effects**: environment mapping, translucency simulated by alpha blending, better shading on the MD2 models, and almost any other effect in RTR3 can be implemented using shaders. A good way to do this is to make your Model return you own PosedModel subclass from its pose() method, and have that PosedModel internally store a Shader and a PosedMD2Model. Ask me about how to used PosedModels before embarking on this—it is relatively simple, but the design pattern can be confusing the first time you encounter it.

4. **Artificial intelligence**: The world will be more interesting if other characters or objects move somewhat intelligently. It is common to model a simple thought process for characters using a state machine. To find their way around the world, most AI characters use the A* path-finding algorithm on a fairly coarse graph. You can create such a graph manually by placing nodes at the center of “rooms” in your level and connecting them by edges. Quake maps already come with a file containing this kind of information. Although G3D does not load it for you, you could find the format on the internet and load it yourself.
5. Top view/target clicking: For an **RTS** or other more **tactical** kind of gameplay, put the camera above the whole scene and look down. Instead of directly controlling the characters’ positions using arrow keys, click on target locations and use path finding to have them move on their own. For multiple characters, make some way of showing them selected (e.g., changing their textures or drawing an **alpha-blended** glow around them).

6. Changing models: To make a character drive a vehicle, simply replace the character model with the vehicle model and remove the “unoccupied” vehicle entity from the world. That is, a character entering a vehicle (or riding a mount) **becomes** the vehicle. When the character dismounts, he or she returns to the original model and physics constants and the “unoccupied” vehicle reappears as a power-up entity in the world. You can also have a single character that can magically transform into a different state. For example, most characters have multiple textures that you could switch between to simulate different states, or in some cases there are even different versions of the model available (e.g., ForgottenOne vs. ForgottenOne2.)

7. Ground alignment: Humanoid characters should stay upright as they move about the world. If you choose to use vehicles, they should tilt based on the surface underneath them (e.g., you might want to load the race car models as very small objects like toys and then race them around a giant world). You can find that surface using ray casts. For cars, it is common to perform one **ray-cast** downward from each wheel and then use those results to align the object. You can even hack up some simple physics based on this so that cars act more like cars and less like humans.

8. Better art: It is hard to make 3D models, but easy to draw texture maps. Most of the maps have decent textures, but they might not fit your theme. A pk3 file is just a zipfile with a different extension. Unzip them and draw on the walls! Most of the MD3 characters have very low-resolution texture maps. Use G3D::GImage to load the PCX files and save them out as JPG or PNG. Then resize and draw on them, either to improve their resolution or to change the theme. There are a lot more Quake 3 maps available for download on the web. There are a many other MD2 models, but G3D comes with most of the good ones. The G3D data tab in the documentation shows **some** of the MD2 models and fonts in G3D that you can use to get started.

9. Cinematic or loose-follow cameras: A follow camera that rigidly sits a certain distance from a character is not always the best viewpoint. Many games have third-person cameras that loosely swing behind the character and that also move closer or farther to avoid having objects come between the camera and the character. Quest-based games like Indigo Prophecy (and the spectator mode in most first-person shooters) also use fixed cameras in the environment and switch between them, as is done in movies.

I recommend that you avoid real physics, audio, and networking for this project because they generally require more time to debug than will be available to you. They are however good features to add if you continue this project for a final project.

---

3 Design Overview

You are starting with a set of classes for a mostly-complete **design** of the game engine. The implementation is partly complete: I’m giving you implementations for pieces where the difficulty of debugging them outweighs the educational benefit of implementing them yourself.
For other pieces you must complete the implementation yourself. In a few places, the design is also left to you. You are not required to use the design that I’m providing, and you can change it wherever you feel necessary to follow your group’s vision.

The basic structure of a game engine is shown in the following diagram:

```
In the provided design, you’ll obtain the level and model data directly from the G3D data directories. The loaders are G3D::ArticulatedModel, G3D::MD2Model, G3D::BSPMap, and G3D::Texture. You have no audio or networking. Each box in the diagram corresponds roughly to a App method: onUserInput, onLogic, onGraphics, onSimulation. In practice, except for onGraphics, most of your code will go in the similarly named methods on the Entity class and not directly on your App. The reason for this is described in the App and Entity documentation and comments.

Every object in the world is represented by an Entity. This maintains its state, including position, velocity, heading, and current animation parameters. The Entity’s geometry is described by a Model. There can be multiple Entities sharing a single model, a practice called instancing that reduces the memory overhead for scenes with many objects. During the rendering process, each Entity is posed, which reduces it to a set of G3D::PosedModels (this happens during the App/Entity onPose method). A PosedModel contains only the information needed for rendering, not additional state about the source Entity. G3D::PosedModel::sortAndRender draws a set of models, so you don’t have to explicitly implement rendering for them.

Level geometry is stored in the Level class, which internally contains a G3D::BSPMap. BSPMaps have a special algorithm for culling geometry that is not visible to the viewer, so they
render themselves directly rather than using the PosedModel architecture. The BSPMap does not contain sufficient information for conveniently applying physics against the level geometry, so Level contains an explicit BSP tree of triangles as well.

### 3.1 Debugging Tools

Here are some tips for debugging. One of the key distinctions in this project is between simulation time and wall-clock time. All game physics should use simulation time (in onSimulation). This ensures that your code operates the same no matter how fast it is actually executing. That is very important if you are in a debugger and single-stepping through a collision routine—you don’t want the game to thing five minutes have elapsed between time steps in that case. If you program carefully, you can use this to alter the simulation time step for either debugging purposes or for in-game special effects (like “bullet-time”). For debugging, you can even set the time step to zero, which freezes the world and lets you examine its state while the rest of your program is still alive.

G3D writes messages to a log.txt file stored in build directory next to your program’s binary. This contains errors that may not appear on the screen and various warnings. You can write to it yourself using logPrintf. This is handy because debugPrintfs that execute immediately before a crash may not actually appear in the xterm before the program crashes (especially if the crash takes out the operating system).

For information that constantly updates, like the position of the player, use screenPrintf. This writes text directly to the screen. You won’t be able to see one-off messages flash past (use debugPrintf for those), but you will see messages printed every frame appear in a stable on-screen position instead of scrolling past in an xterm.

If you’re getting OpenGL errors, add debugAssertGLOk calls to your code to try and pinpoint the routine that is causing them. Often one routine will create the underlying error but the program won’t fail until much later.

The 3D equivalent of screenPrintf is debugDraw. This lets you add G3D::Shape subclasses to a list of objects to be drawn each frame. Look at how I use this in the Level::slideMove starter code.

You’re going to write a lot of classes this week. G3D contains a script called “newclass” that produces a matching .h and .cpp file for you. This speeds up the writing of many classes. E.g., “newclass –r PowerUp” writes PowerUp.h and PowerUp.cpp and makes PowerUp a reference counted object.

All GApps contain a console, just like most games. The console lets you augment your debugging GUI. You can access it by pressing “~” (shift-quote) and close it by pressing ESC. You can also set the default GApp ESC behavior to open the console. You can print to the console using consolePrintf, which is a nice place for debugging statements that you don’t want the eventual player to see but do want to keep in the program (e.g., “Collision between Player and Missile occurred”). More significantly, you can parse the command line input and use it to drive your program. For example, in App::onConsoleCommand you could recognize “spawn classname” and have it trigger creation of a new classname Entity 5m in front of the camera. If you press “~” in Half-Life, Doom, or Quake you’ll find that they all have this functionality. See
the TextInput documentation for more information on parsing. Consider other console commands that would be really helpful for debugging…

## 4 Tasks & Advice

I explicitly marked many of the places where you need to add code with Begin TODO…End TODO comments. I can’t mark all of the places you need to touch because what you change depends on how you modify and extend the base design.

The major tasks and pieces of functionality are:

1. **Model**: mostly requires reading (lots of) G3D documentation and creating an elegant design for readability
2. **Entity** and its subclasses: little code for the engine but lots of tricky logic and physics to think about for the game itself.
3. **Level::intersect** and **slideMove** methods: lots of geometry code similar to the ray-tracer and reading the G3D::CollisionDetection docs.
4. **Level::renderShadows**: lots of GPU manipulation, texture coordinate manipulation, and reading RenderDevice/OpenGL specifications
5. **App**: some game logic that doesn’t belong in any one Entity, potentially extensive 2D code for the in-game HUD (heads-up-display/on-screen score and other stats) and debugging GUIs, and a small amount of straightforward code for setting up the scene(s).
6. **Art**: You’ll need to construct more complicated scenes and animations than you ever have before, and will likely want to manipulate some of the texture maps (and possibly download new models from the internet). Plan art time into your schedule and task assignment.

AI: Not required for part A, but for part B, writing a **pathfinding**, **flocking**, or other AI algorithm is probably more challenging than all of the Level intersect methods combined.

These can be divided among team members. Each is about the same amount of work, but that work will come at different stages in the process. They are needed roughly in the order specified above, although you’ll have to stub out each other’s implementations in the beginning to test your own.

It is a good idea to have a single person in charge of managing the team and give them reduced coding responsibilities. This person should resolve architectural disputes, organize group meeting times, ensure that the documentation (especially known bugs!) is up to date, and ensure that the project is proceeding on schedule.

Because I’m giving you the public interfaces for most of the classes, you can work independently at first. Feel free to change those interfaces, but be aware of whose work you’re impacting. Try to break dependencies between each other. For example, the Entity programmer can work on physics and game logic without models by simply rendering cubes instead of the characters.

### 4.1 Working with SVN

For a project with a lot of people, a good discipline to establish is that you should (generally) not check in code that would “break the build” by preventing others from compiling or running. All this means is that you should use `#if 0…#endif` or block `/* */` comments to turn off broken code before checking in so that you don’t slow down your teammates.
Some counterparts of this discipline that will help keep you out of trouble are “best practices” for revision control on a team:

1. E-mail your team whenever you check in code to let them know the current status.
2. Keep your documentation up-to-date at all times, especially with notes about what is not working and known bugs.
3. svn update regularly, and svn commit whenever you’ve completed a working piece of functionality even if you’re going to keep working that session.
4. Build and run the game first thing when you sit down, so that you’ll know if someone else checked in broken code. If they did, use “svn log filename” to see who changed what when, and then let the perpetrator know that they made a mistake that is costing you time!
5. If you’re going to have to check in broken code (e.g., to get it to someone else’s check out so they can help debug), e-mail your group immediately. This will let the team know that they should avoid updating until you’ve fixed it. Often you can use #if to check in the code without it breaking the build.

### 4.2 App

The App class is largely stubbed out to get you going. I set it up with some extra #ifdef statements to support Windows and OS X and included a game.sln file for those of you working on different operating systems.

I put the camera movement code in App::onSimulation to get you started, but this is a lousy place for it. A better design is to make Cameras into special Entity subclasses that contain a GCamera and do not undergo normal physics or collisions. A Camera can rigidly track an object or move more fluidly. I intentionally did not give you a follow-camera to satisfy the specification—at a minimum, you must figure out how to make it swing to stay behind the character.

Remember that you can use the F2 key to switch between the in game and debugging cameras. It helps if you add a checkbox to draw your Camera entity (perhaps using debugDraw) so that you can figure out where it is when looking from the free camera.

Once you’ve built your engine, you will probably want to make some major changes to App to implement your game. These include a HUD or player GUI, game logic that isn’t specific to one particular Entity, and any data that needs to be shared between objects in the world.

### 4.3 Entity

There isn’t a lot in Entity, the base class. I implemented the Entity-world physics but left the Entity-Entity physics for you. There you should detect when one Entity’s bounding sphere is going to hit another’s and resolve that collision in some useful way. You could simply set sdt to be the tie at which they will hit, so that the Entity moves up to the other one and stops. Or you could advance to the collision point, perform a momentum exchange as governed by Newton’s laws, and then continue the simulation (making characters act like billiard balls). Or you could have one Entity push the other one. It is up to you.

Even in Entity and its subclasses, always use setPosition/setHeading instead of mutating the underlying variables directly, so that you can ensure that any dependent data is always up to date.
The Player class will initially be your debugging character to run around the map. In onUserInput or onEvent, respond to keyboard events in a way that changes the character’s velocity. Take advantage of the touchingGround method to restrict certain actions to cases where they character is not falling. For an MD2 model, you can drive the animation by setting the m_action fields as appropriate. The MD2 model animation will automatically compute the right animation to play when conflicting fields are set.

Once you’ve got the basic engine down, there’s a lot of work to be done in Entity and the subclasses. This is where all of the game rules are. When a character collides with a power-up, remove the power-up disappear from the world and apply its effects to the Entity. When a character throws a mushroom, add a mushroom Entity to the world with some high initial velocity. When a rocket hits a wall, remove the rocket entity, spawn a particle system in its place, and apply damage and outgoing velocity to all other Entities nearby.

Don’t feel constrained by the initial physics. You can make characters fly, double-jump off the air, run up walls, or any other features needed to realize your game design vision.

4.4 Model

The model class separates geometric descriptions from posed geometry and from individual objects in the world. It doesn’t contain any interesting algorithms but must be carefully designed to make it maintainable. It is easy to make a very bad model implementation that contains lots of duplicated code.

The first design challenge in Model is that it must abstract over both ArticulatedModel (for IFS, OFF, 3DS, and PLY2 models) and MD2Model (for Quake 2 MD2 modes), which have very different APIs.

MD2Model does not contain its own texture and color information but instead expects those to be provided at pose time. So you must maintain that information for the MD2 model inside your own model. MD2 models are in two pieces: the tris.md2 character and the weapon.md2 “weapon”. The weapon is actually not a weapon in all cases, but might just be a second piece of the character. So you always need to load both. The weapon texture is always weapon.pcx. The character texture you usually have a choice of. The PCX files are a very old file format that is supported by Texture and GImage but by very few modern paint programs. The PCX textures used for quake are too dark, so you need to brighten them by a factor of 2 and gamma correct them by about 1.6. Both of those are supported by the Texture::PreProcess argument to the Texture::FromFile method. Only apply those to PCX files, so that if you load a .jpg file that you drew yourself you won’t get the extra post-processing. Quake models are also too big by default. Scale them down by about 0.5f. I provide an extra “scale” argument to my constructors that is applied on top of this for the cases where I want to make a very tiny or very large model intentionally.
The Pose class contains pose information for both MD2Model and ArticulatedModel. To animate an articulated model, explicitly set the coordinate frames for each of its named Parts in the Table. To animate an MD2Model, modify the action used in Entity to simulate the model.

Both ArticulatedModel and MD2Model produce their own PosedModel subclasses. Look at the G3D source code if you are interested in seeing how these are implemented. The ArticulatedModel one supports full G3D::Materials, which use Phong BSDFs, support many point or directional lights, and even have support for relief mapping, alpha transparency and environment map reflections. MD2 models aren’t quite as rich—they have a simple per-vertex Phong model. If you would like to add more features to MD2Model, you can write a PosedModel wrapper that takes the PosedMD2Model and surrounds it with your own Shader code. The PosedModel API is a little complicated, but there are many examples of its use in the G3D source code.

4.5 Levels

Quake3 levels are stored in pk3 files. These are just zipfiles with a funny extension. You can rename the file and unzip them to see the contents. G3D can work with files in a zipfile or directly on the file system. To read inside a file, just use the filename as if it was a directory and keep adding slashes and filenames after it.

In the pk3 file, there are several subdirectories. The “maps” subdirectory contains one or more “.bsp” files that contain the actual geometry. The “textures” directory contains some of the textures used. Other textures are stored in a common pak0.pk3 file that ships with the actual game. I put that file in G3D/data/quake3 for you and G3D will locate it by default. Note that this file is not redistributable, so if you want to distribute your project later you’ll either have to redraw some of the textures from this file, choose maps that don’t use them, or put up with textures not appearing in your game.
Some other directories and files contain information useful for implementing AI characters, for sound, and for browsing maps. You are welcome to use those files but are not required and I don’t know their formats, so you’ll have to rely on internet resources (the entire Quake3 source code and several clones are available, so there are a lot of resources).

I provided the code to load the map and store the triangles in a BSP tree that you can use for the intersect methods. I also wrote the code to render the map. Note that some textures won’t be in either the pak0.pk3 file or the pk3 file for your specific model. E.g., the wall and column textures for the image on the previous page simply aren’t in the Quake distribution due to a model error. You can let G3D choose a default texture for you or specify the explicit BSPMap default texture to use in these cases.

### 4.6 Level intersection methods

3D physics involves computing the locations at which moving objects strike one another. If one object is a point moving with constant velocity relative to another object (i.e, the point moves along a ray), the location of impact is given by a ray intersection, just like in ray tracing. When the objects are more complicated, the intersection computations become more complicated, but they are still given by the mathematical intersection of the shapes.

Level supports several intersection methods that are used for physics and game logic. These are:

1. `intersectMovingSphere`
2. `intersectMovingPoint`
3. `intersectBox`
4. `intersectSphere`
5. `getSplashTris`

These have extensive documentation. The starter code contains the BSP tree that you’ll need for them and most of the setup for the ray-triangle intersection. You have to do the rest, which is marked with TODO. Use your ray tracer’s triangle intersection code and the methods on the G3D::`CollisionDetection` class to help you.

When debugging these methods, use `debugDraw` to visualize the hit location, the hit triangles, and the ray, sphere, or box geometry. It is useful to use a top-level Boolean variable to turn on and off your debugging code so that you can access it from the GUI. You’ll find yourself returning to the debugging code frequently even after the intersect methods are “done”, since everything else in the game depends on them.

In addition to the applications that are explicitly required, you can use these intersection tests for:

1. Finding whether one object can see another, which is particularly useful for AI
2. Finding all objects within an area of another, which is useful for area-effects like healing and transportation spells, explosion impacts, and the effects of gas clouds
3. Determining the first object struck by a very fast projectile, like a bullet or fired grappling hook. Fast projectiles are not simulated by Entities but instead their effects are instantaneously applied. These are called `insta-hit` or `hit-scan` in many FPS games.
4.7 Level::renderShadows

Most new 3D games use shadow maps to render real-time shadows from dynamic light sources in real time. You can read about this algorithm in RTR3, and it is implemented in ArticulatedModel / SuperShader. You will not use that algorithm for this project because shadow maps do not interact correctly with precomputed illumination in the light map. Instead, you will render drop shadows. These look like the dark circle for the image on the right.

The algorithm for drop shadow rendering for one Entity is:

1. Find the Level triangles underneath the Entity
2. Project a dark circle texture onto those triangles
3. Darken the triangles by the texture

To find the triangles underneath the Entity, use getSplashTris. Make sure that getSplashTris works correctly before moving on. A good way to do this is to use debugDraw to draw those triangles from within renderShadows.

In order to project a texture onto the scene, you need to compute new texture coordinates for the Level triangles such that (0.5, 0.5) corresponds to the location directly under the Entity and that the unit square maps to a region with the same extents in x and z as the Entity’s bounding sphere. The algorithm for doing this only requires one or two lines to implement. Note that you can ignore y values entirely. To see if you’ve got the right texture coordinates, draw those triangles (yourself, using RenderDevice::begin, not using debugDraw) and set the color equal to the texture coordinate like you did when debugging the ray tracer. You should see something like this when you’ve got it right:

Now you’re ready to actually project an image onto the ground. Begin by putting the entire body of your renderShadows method into RenderDevice::pushState…popState so that you leave the GPU state the same as you found it to avoid corrupting other methods that execute later.
I already loaded the texture for you in Level::Level(). There are two textures: debug-shadow.tga and shadow.tga. Use the debug version until your code is working. This texture is loaded with WrapMode::CLAMP, which causes texture coordinates to be clamped to the unit square (by default, they wrap using a floating-point modulo operation). This means that even very long triangles will only receive shadows just under the character, and not tiled across their surface. To render using this texture, configure your render device as follows (see the RenderDevice documentation for details; I want you to look them up so that you get familiar with the docs):

1. Linear interpolation **blending function** of $\alpha \star \text{src} + (1 - \alpha) \star \text{dst}$
2. **Depth test** of equal, so that you only overdraw existing triangles
3. Disable lighting (this turns off some legacy hardware features)
4. Smooth shading mode
5. Set texture unit 0 to use the shadow texture
6. Color of white (the color will come from the texture)
7. $\alpha$-test of $> 0.01$

The $\alpha$ (“alpha” or “A”) values are a fourth, invisible color channel that is stored in the shadow texture with R, G, and B. This channel specifies the per-pixel blending weight. Where $\alpha = 0$, the existing screen image should be used. Where $\alpha = 1$, the texture completely replaces the screen image. Intermediate values specify the fraction of the blend.

If your texture coordinates and state are correct, when you render the debug texture you should see something like the image on the right. When it is working correctly, replace the debug texture with the real one.

This technique is called a “**decal**” or “**projective texture**”, depending on the application. You can use the same method with a different texture to render transient textures like spraypainted logos, footprints, growing ivy, bullet holes, or splashed liquids on the walls and floor.

**5 Groups**

At any point starting now, you can e-mail me a list of your group members and I'll create the SVN project for you to work from. You can wait until after the assignment is posted to decide if you wish. The only rules for group formation are:

1. You have to actually ask the other members of the group before you sign them up!
2. Your group must be named after a fast animal, e.g., arctictern
3. Each person must be in exactly one group
4. Groups may not contain more than four members

Please send me one e-mail per group, listing the group name and the UNIX names of the members as a comma separated list, e.g.,

artictern = 09sb, 10tkr, 09afw

I will then create a SVN project for you that you can access with:

```
svn co svn://graphics-svn.cs.williams.edu/371/4/game-artictern
```