CS 371 Project 7:

Interaction

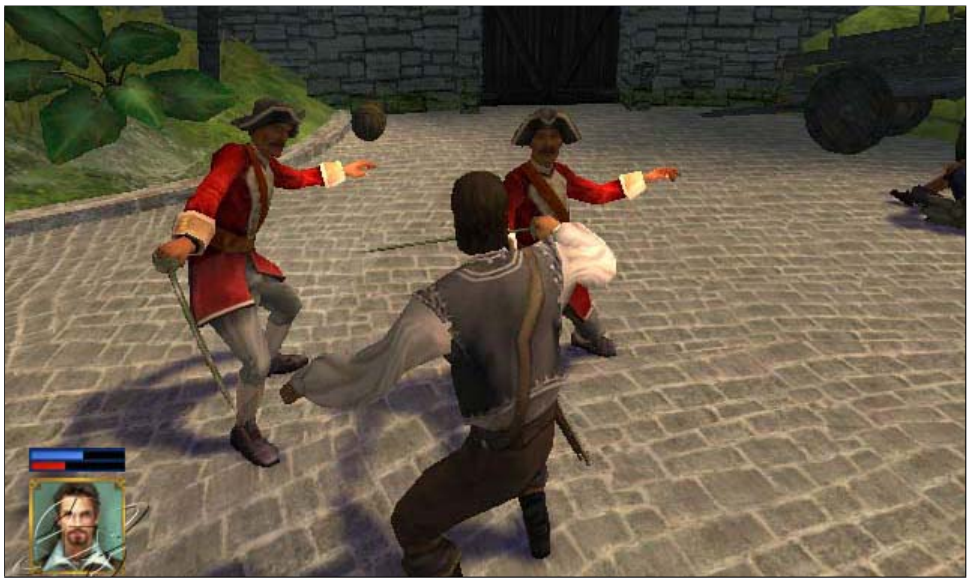

Figure 1: *An interactive world with multiple characters, dynamic effects, and an integrated heads-up-display for a GUI.*

1 Introduction

To immerse the player, the ability to explore and affect a virtual world is as crucial as its appearance. In this project, you'll extend the real-time rendering system from the previous week with features that allow a user to interact with the environment and objects within it through an avatar.

Many students choose to create combat-based video game frameworks (e.g., like *Halo*) for this assignment for the same reason that many game designers have: it is easy to quantify and model the limited interactions in that context. You should not feel limited to that kind of approach, however, and I encourage exploring more interesting kinds of interaction such as physics puzzles (e.g., *Portal*); a creative construction environment (e.g., *Minecraft*); and character relationships based on trading and communication (e.g., *Harvest Moon* and *Skyrim*) or joint navigation (e.g., *Uncharted*, *Prince of Persia*, and *Ico*). If you're considering a real-time or interactive final project, consider exploring and building suitable infrastructure for it in this project.

1.1 Process

The specific features that your program will contain are your choice. This handout gives a prototype specification. Don't implement this specification as written, and

don't start working on the implementation until Tuesday.

I intentionally added more features to the interaction project than I think you can effectively implement in a reasonable (8-12 hours for each member of the team) amount of time. Your job is to negotiate this down to a feature set that has educational value and can produce compelling visuals. You can also add new features or substitute them for the ones here–there may be something you want to learn that is not covered yet.

I will discuss your new specification with you in class next Monday. You are responsible for getting me to approve it before then. You can re-negotiate your specification during the project, but I am more amenable to change before production begins.

Just as on the midterm and final, I will evaluate your work against *your own specification*, considering both how much you did and how well you did it. So it is important that you scope the work well. The only elements not open for adjustment are the deadlines, that the project must build on this week's, and that it must focus on interaction.

1.2 Advice

I suggest that you work backwards from the deliverables. For class projects, those are your **education**, **compelling visual results** that communicate technical correctness, and your **personal satisfaction**. Every feature has some cost in developer time. That cost is further magnified by its risk. The net cost of a feature is balanced by its value towards your deliverables. You want to choose a feature set that maximizes the value produced by your limited development time.

For example, in project 1 (meshes), texture mapping required only a few lines of code, but made your visuals much more impressive and gave you hands-on experience with a core graphics technique. It also shifted complexity out of code and into data. These properties make it a good feature of the assignment. In contrast, I did not assign texture coordinate generation for the cube and cylinder. Those are much more difficult to texture than a heightfield. Adding texture coordinates to two more shapes offered minimal incremental educational value, and would not have made your results look substantially different–the heightfield is more impressive than a tube or a cube. So that was a low-value feature and I cut it from the project.

The "constrained movement" section of the specification contains significant educational value and is very satisfying to implement. It is also the part that will push you the farthest from elements that we've already studied. Plan it carefully and consider working on that section early to understand it better.

1.3 Educational Goals

1. Apply geometric algorithms to interaction and simulation problems such as collision detection and response
2. Practice using the output merger for blending
3. Learn new hardware-accelerated rendering techniques
4. Practice scoping project work

5. Practice parallel team software development

1.4 Schedule

This is an **easy**, pair project that builds on last week's real-time rendering system. The project is only easy if you create a reasonable specification and work effectively as a team on it. So scale your specification accordingly. Plan for *at most* eight hours of work per person on this project once production starts.

Start Preproduction:	Thursday,	November 1	2:30 pm
Specification Due:	Monday,	November 5	12:00 pm
Start Production*:	Wednesday,	November 7	
Checkpoint:	Thursday,	November 8,	1:00 pm
Due:	Monday,	November 12,	**12:00 pm**

Remember to scope your work appropriately and reserve a substantial portion of time for the post-production work of creating the report and images.

Take into account that while you are in production on this project, you will also be in preproduction on your final project.

* You are *permitted* start working before the production start date, but I assume that you'll want to put effort before the start date into project 6, and if you start work before I approve your specification you risk spending time on features for which you won't being credited.

1.5 Rules/Honor Code

You may modify these rules as part of your specification.

Work in a group of 1-4 people. You are encouraged to talk to other students and share strategies and programming techniques. You should not look at any other group's code for this project or use code from other external sources except for the G3D library and code from your textbook. You may use any student's code from the *previous* projects this semester with his or her permission. During this project, you may use any part of the G3D library and look at all of its source code, including sample programs.

For this project, I want you to primarily work in parallel with your teammates instead of pair programming when writing new code. You should commit code to SVN under your own account. Working effectively in this environment requires careful planning and division of labor, as well as code organization–avoid merge conflicts and broken builds by planning the development process. You may debug and work on postproduction (scenes and report) by pair programming.

You may share data files and can collaborate with other groups to create test and visually impressive scenes. If you share a visually impressive scene with another group, ensure that you use different camera angles or make other modifications to distinguish your image of it.

You may copy and paste from this document's electronic form to form all or portions of your specification.

After Monday, Nov. 5 at 12 pm you may not modify the `specification.dox` file or image files related to it without my prior approval for your specific edits.

2 Specification

Do not implement this specification in the form that it appears below. Modify it and submit the result to me as your proposal. You must format your proposed specification as a detailed, numbered nested list in Doxygen, saved as `specification.dox` in your root SVN directory. The nested numbering is for reference when grading. Each numbered item should therefore correspond to one explicit task that I can check for correctness and completeness. Compare your previous project evaluations and to the specifications for an example of how that works.

You must create your own checkpoint definition and report specification as well. When writing your report, be sure to give evidence that you have completed each task from your specification. The best way to do this is to include images that could not have been produced without that task operating (mostly) correctly and leave Doxygen links explaining where in your codebase I can find the corresponding source.

This specification has more detail than the typical ones because it incorporates information that I usually reserve for implementation advice as well as being too ambitious for one week of work. Look up the documentation, think through the implementation *and* debugging *and* presentation, and then decide what is reasonable.

For each numbered element of the specification, note which team member will be responsible for implementing it. List them in the order that you plan to implement the features.

1. **Dynamic Entitys**
   (a) Create a new `DynamicEntity` subclass of `VisibleEntity` that uses its own logic instead of `PhysicsFrameSpline` for motion and animation.
   (b) Create a new subclass of `Scene` that to recognizes and instantiate `DynamicEntity`s appropriately.
   (c) Create a `TriTree` named `m_collisionTree` inside the scene and populate it from the non-dynamic (i.e., static) `Entity`s, as if you were going to ray trace them.
   (d) Create a `PlayerEntity` subclass of `DynamicEntity` that has state for its *desired* velocity and orientation.
      i. In its simulation method, create an actual velocity and orientation from these subject to maximum acceleration constraints, and then update the position from them.
      ii. Extend your testing scene with a `PlayerEntity` that uses an MD2 model.
      iii. Use the `App::onUserInput` method to set the desired velocity and heading based on keyboard input. You may want to directly extract information from the `FirstPersonManipulator` when the debug camera is disabled (by pushing the F2 key). You should now have a character that can fly around.

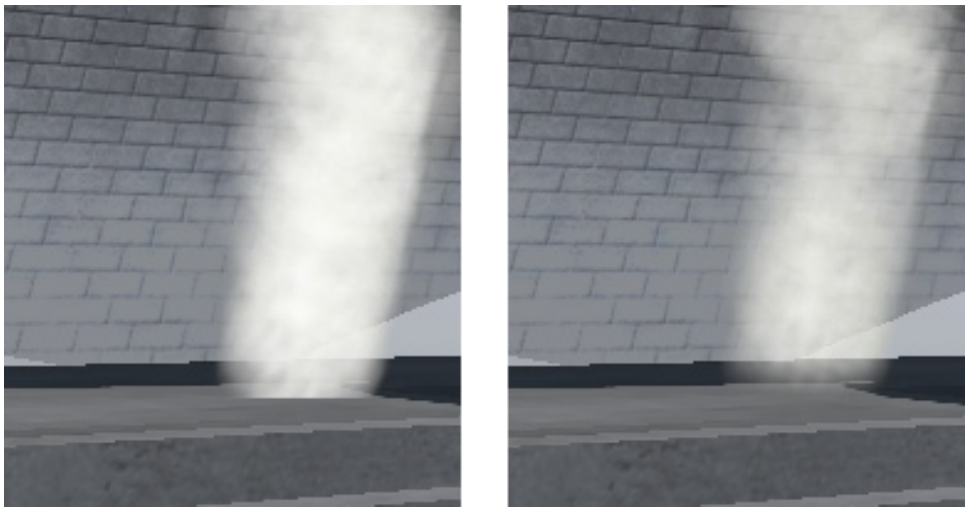

Figure 2: *Left: regular particles reveal their 2D nature at intersections with the ground. Right: soft particles fade out coverage near intersections.*

2. **Particle System**

   (a) Create a `ParticleSystem` class that supports spawning and of point particles (you'll need to do some research and look in the textbooks for more information) with constant velocity. Do not make this an `Entity`.

   (b) Create a debugging visualization of your particle system that draws each particle as a point.

   (c) Add the particle system to your `App` (not your `Scene` subclass) and invoke its simulate and render methods. Be sure to render *after* everything else in the scene, and do not write to the depth buffer. Pass the depth buffer and light (...and shadow map) into the render method for later use.

   (d) Extend your particle system to render in a more interesting way by creating a rectangular billboard of some fixed size around each point that faces towards the camera. You can perform the point-to-billboard computation on either the CPU or the GPU, but should choose one for your own specification.

   (e) In `App::onUserInput`, create smoke 3m in front of the camera along its look vector whenever then spacebar is held down. You should be able to create nice plumes of smoke with this.

   (f) Extend `ParticleSystem.vrt` and `ParticleSystem.pix` with texture map. The `.a` channel of the texture becomes the `.a` channel of `gl_FragData[0]`, which controls the alpha blending.

   (g) Add lighting and shadows to the particles (particles receive, but do not cast shadows) as if they were visible surfaces. Note that the particle has no natural "normal", so you need to devise a solution for shading.

(h) The particles create a line wherever they intersect geometry, revealing their flat 2D nature (Fig. 2. To disguise this, implement the **soft particle** algorithm:

i. Read the depth buffer value at texel position

```
vec2 c = gl_FragCoord.xy * g3d_sampler2DInvSize(depthBuffer);
```

This is a number on the range [0, 1] that is the z/w value for the post-division, post-projective point.

ii. Compare the depth buffer value to `gl_FragCoord.z`, which is the particle's z/w value. Where they are close, the particle is about to enter a surface, so reduce its alpha value. I chose to just scale their absolute distance by 20, clamp it to [0, 1], and modulate the alpha channel by this "hardness" value.

3. **Constrained Movement**

Objects should be constrained by gravity and by collisions. Gravity is trivial to model: if an object is not touching the ground, set its desired velocity to be negative along the $\hat{y}$ axis. For collisions, you must find the geometry that an object will intersect if it moves by its desired velocity and select an appropriate actual velocity that prevents the collision.

(a) Implement methods on Scene to find the triangles that intersect:

i. A moving ball (sphere)

ii. A static ball (sphere)

iii. A static (axis-aligned) box

iv. A ray, using a `Tri::Intersector`

(b) Abstract each dynamic entity as a bounding sphere. Constrain their motion so that they can only move slightly less distance than would cause that sphere to interpenetrate triangles in the world that are detected efficiently using the methods from the previous step.

Tip: Look at `G3D::CollisionDetection` in addition to methods on individual geometric primitive classes. You *may* find the `G3D::BSPMap` source code useful, although it can be confusing.

(c) Extend your simulation to make dynamic entities slide along triangles that they collide with rather than stopping abruptly. You will need this to make objects able to move along the floor once gravity is added, but it also dramatically improves the feel when moving near a wall. Note that you will have to make your collision response system iterative, since responding to the first collision changes an object's motion, and that can change other collisions that will happen in the same time frame. There are two common methods for sliding: one cancels all velocity in any direction that would cause a collision. The other cancels that velocity, but then increases velocity in other directions so that the magnitude of the velocity vector is preserved. Choose one and explain why.

(d) Add gravity to your simulation.

(e) Prevent collisions between dynamic entities (still treating them as spheres).

4. **Hit-Scan and Decals**
   Hit scan weapons in games instantly determine their hit position by a ray trace, rather than creating a projectile. These were initially introduced for laser weapons, but are today typically used to model bullets, which would reach their target before the next frame was simulated.

   (a) Create the ability to "fire" a weapon by tracing a ray from the player character into the world when a key is pressed. Use `debugDraw` to display a sphere at that location while the firing key is held down for debugging purposes. This ray cast should only involve the `TriTree` of static objects.

   (b) Draw the laser beam between the player and the hit location. This can be done using `Draw` calls or an explicit `RenderDevice::apply` call, but do not create a new Entity.

   (c) Spawn new particles at the hit location, as if the laser were burning into the wall. Optionally, maintain a list of hit locations and continually spawn particles at each of them for a short period of time so that the burning continues after the beam is turned off.

   (d) To further mark the impact, games typically overlay **decals** that give the appearance of changing the world texture without actually modifying it as shown in Figure 3. Create a `DecalSystem` that is like the particle system in maintaining a number of small objects, but since it doesn't need to animate them can be much simpler and assume a single `Texture` is applied to all decals. The decal system should operate as follows:

      i. Find all static triangles within a small radius of the intersection point, i.e., find the intersection of a ball (solid sphere) and the world.

      ii. Add those triangles to the decal list. Assign new texture coordinates to them based on the distance from the point intersected by the original hit scan *in the plane of the triangle intersected by the hit scan*. You will need to construct an (arbitrary) set of orthogonal axes in that plane to correspond to the $\hat{u}$ and $\hat{v}$ axes of your texture space.

      iii. When rendering, shade the decals as if they were visible geometry. Note that you can re-use your direct illumination pixel shader (don't even make a copy of the file!), provided it handles the alpha channel properly. Remember to turn on "$\alpha, 1-\alpha$" blending on the `RenderDevice` or you will see black in the $\alpha = 0$ areas.

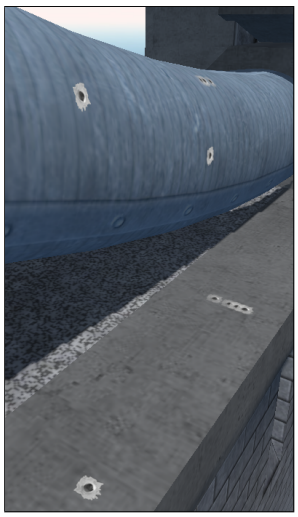

Figure 3: *Bullet holes rendered as decals on curved surfaces and geometry with varying orientations.*

5. **Ambient Occlusion Drop Shadows**
   Create a system to draw drop shadows under dynamic entities to simulate ambient occlusion and make their position clear when they are entirely in shadow. This is almost identical to the `DecalSystem`, except that the decals move every frame. Find all triangles under an entity in a radius proportional

Tip: I debug the texture coordinates for decals and drop shadows by using them as the color, like this:

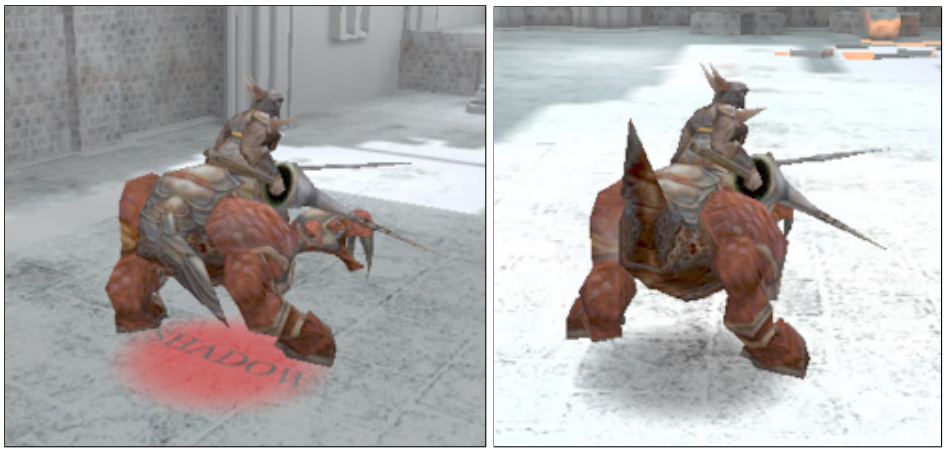

Figure 4: *Testing the drop shadow with a temporary texture, and the final result.*

to the size of the entity, create some proxy geometry, and apply a dark decal to it. Remember to update these every frame!

6. **HUD and Inventory**

   (a) Add state to the player for health, points, and four tool slots. Three of the tool slots should initially be empty and the first should initially hold the hit-scan laser tool.

   (b) Display the health and points state on screen in `App::onGraphics2D` using `GFont::draw2D`. You may augment this with bars or replace it with icons.

   (c) Display the currently available tools. You can either use `RenderDevice::setViewport` to create a small viewport and render them in 3D as part of the inventory or use 2D icons with `Draw::fastRect2D`.

   (d) Implement tool switching. Use the numbers 1-4 on the keyboard to select different tools, and highlight the currently active tool.

   (e) Targeting is hard in a 3rd person view. To improve this, continuously cast rays forward from the player character and display a 2D crosshair at the corresponding intersection point in 3D. `GCamera::project` implements the 3D to 2D projection you need.

7. **Items**

   (a) Create a new `ItemEntity` subclass of `DynamicEntity`. These are objects that can be picked up and dropped by the player.

   (b) Extend the scene description file to allow instantiating `ItemEntity`s with varying properties. For example, one might be a health vial and another might be a new tool. I chose to create a small subclass hierarchy for this.

   (c) Detect collisions between the bounding box of the player and the bounding box of an `ItemEntity`. When this occurs, remove the `ItemEntity` from the scene and update the player's state accordingly.

(d) Distinguish the multiple tool items in their use. Be creative: you can make tools like a jetpack, medical pack, various kinds of non-hitscan range and melee weapons, night vision, etc. At this point your system has enough complexity that adding features with this kind of impact should take little code (although the debugging and testing might still be substantive).