CS 371 Project 6:

Real-Time Graphics

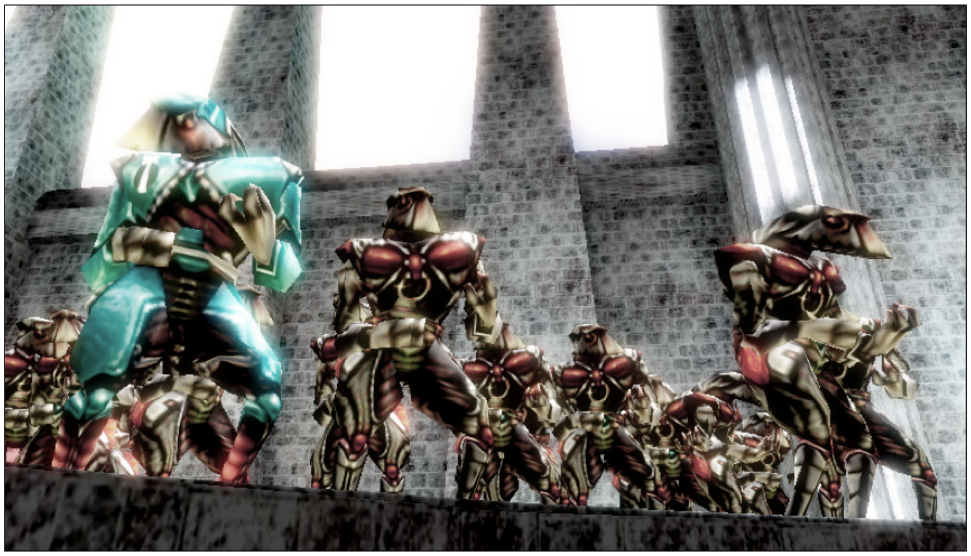

Figure 1: *Video game scene rendered at 60 fps on the GPU with dynamic lighting and shadows using vertex and pixel shaders.*

1 Introduction

Realistic video game graphics must deliver an approximation of physics within a hard time budget of 15 to 35 ms. The rasterization algorithm and massive concurrency in the graphics pipeline makes this possible. Adapting physically based rendering from a ray tracing context to this hardware rasterization context means rethinking the algorithms and tradeoffs within them. It also requires learning to work with low-level tasks like GPU memory management, run-time compilation, and managing graphics card state. In this project you'll do all of these, building a complete rendering system suitable for a video game given the model loading and application infrastructure that we've built throughout the semester.

In addition to creating a real-time renderer for next Monday's deadline, you're also going to create a specification for the following week's project. This is an opportunity to practice what you learned about scheduling and designing during the midterm project.

1.1 Educational Goals

1. Apply physically-based rendering in a hardware rasterization context
   (a) Vertex shader transformation from object to world to camera to projective screen space
   (b) Pixel shader BSDF implementation
   (c) Light visibility determination via the shadow mapping algorithm [Williams 1978]
   (d) Gain experience developing software for a constrained, embedded environment
   (e) Work with functional, massively-concurrent, pipelined programming
2. Analyzing and improving the development process
   (a) Scoping and managing complexity
   (b) Effective visual communication of results

1.2 Schedule

Out:	Wednesday,	October 31	
Checkpoint (sec. 2.1):	Thursday,	November 1,	2:30 pm
Specification Exercise (sec. 2.2):	Monday,	November 5,	12:00 pm (in class)
Due:	Monday,	November 5,	12:00 pm (SVN)

This is a moderately challenging, **individual** project that builds on last week's GPU-programming tutorial. It will form the basis for next week's interactive graphics project. As a reference, my implementation contained 9 files, 420 statements, and 300 comment lines as reported by iCompile.

You've already seen that debugging in the presence of the multiple languages, flakey compilers, and poor specifications associated with OpenGL and GLSL can be time consuming. That's part of life as a graphics (or any systems...) programmer, and it is the same under pretty much every graphics development environment.

This is where it is important to employ the software engineering practices that you learned earlier in the course. Minimize state. Make your program look as much like the math as possible. Test frequently. Commit after every major step. Use assertions. Read the documentation carefully. Look at the source code inside library routines. Avoid language features that you don't understand.

Once you get your programming running, you still have to debug it. You aren't going to be able to print or even run the debugger on the GPU. So you have to devise ways of testing your hypotheses about what is wrong with the program using only one color per pixel as your output. See the debugging chapter of Shirley et al. for a good discussion of graphics debugging techniques.

Beware that the shadow map implementation will require the fewest lines of code of any element of the specification, but requires the most thought and is nearly impossible to debug unless you understand the math. Plan accordingly! There's no reason you have to complete the specification in the order that I wrote it.

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

1.3 Rules/Honor Code

You are encouraged to talk to other students and share strategies and programming techniques. You should not look at any other student's code for this project or use code from other external sources except for the G3D library and code from your textbooks.

During this project, you may use any part of the G3D library and look at all of its source code, including sample programs. I encourage you to look at the shader sample programs and the implementation of UniversalSurface in the `data-files/UniversalSurface/UniversalSurface_*` files (although beware that they are much more complicated than what you have to implement!)

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

After Monday, Nov. 5 at 12 pm, you may not modify the `specification.dox` file or image files related to it in your project 7 directory without my prior approval for your specific edits.

2 Specification

By this point in the course, you don't need me to tell you how to write a report. Provide one for this project as usual, emphasizing the results that demonstrate correctness (since there are few design decisions required). You needn't include a 'feedback' section this week–everyone's workflow is pretty solid now.

Tip: All of the equations that you need this week are in the Graphics Codex, including shadow map pseudocode.

1. Create a scene with a Quake 3 level, a spot light, and an appropriate sky cube map.

2. Implement object-to-world transformation and perspective projection in a vertex shader, and show a normal-vector visualization as evidence of correctness, e.g., Figure 3.

3. Model the sun as a shadow-casting **spot** light in a pixel shader, and show a shading-only image (i.e., no texture) as evidence of correctness, e.g., Figure 5.

4. Implement a Lambertian BSDF in a pixel shader, and show a texture only (no lighting) image as evidence of correctness, e.g., Figure 4.

5. Implement environment map diffuse lighting, and show before and after images as evidence of correctness, e.g., Figures 6 and 7.

6. Implement shadow mapping and show an image as evidence of correctness, e.g., Figure 8.

7. Decorate your scene with animated character models (e.g., MD2 models) and other non-animated props, and render a fly-through video using a smooth camera spline. The video must be relatively short and less than 2 MB for SVN to allow you to submit it. Try targeting 640×480 for at most 20 seconds.

Tip: Read the whole specification, ask me about parts that you don't understand right away, and try answering the report questions *before* you write significant amounts of code.

8. Answer the following questions. As always, you'll get the most out of the lab if you think about these deeply. They primarily are chosen to lead you to interesting conclusions, not to test your knowledge.

   (a) Explain the algorithm implemented by Listing 3.3. Why does the result produce lighting comparable to treating every sky pixel as a light source if it uses only a small number of directions (make an argument with mathematics, starting from the rendering equation)? What is the `max` expression for in each line? Why did I use MIP level 9.0?

   (b) Explain why switching to *front face* culling for the shadow map gives correct results at all, and why it also allows us to reduce the bias magnitude compared to back face culling.

(c) Describe how you would extend the system to handle multiple lights reasonably efficiently. Note that the current system assumes that all lights affect all objects, and computes bounds for the Quake 3 levels in a relatively ineffective way under spot lights.

(d) Explain in a mathematically rigorous way why you can perform the shadow map projection *before* interpolation, in the vertex stage, and get the same result as if you projected in the pixel shader after interpolation of position across the triangle.

Like the example images in the following section, all of your images should have the same aspect ratio, be from the same viewpoint, and be uncluttered by the GUI. Choose your viewpoint so that it illustrates each of the features well, and use your knowledge of art composition to make the images interesting.

In addition, you must submit the checkpoint report, post-mortem evaluation on the midterm, and project 6 specification exercise described in the subsequent sections.

Your repository directory for this project is:

`svn://graphics-svn.cs.williams.edu/371/6-RealTime/realtime-<username>`

2.1 Checkpoint **(due Thursday, Nov. 1, 2:30 pm)**

1. Create a report draft, including placeholder images and a very tiny sample video of a fly-through of a Quake scene *that was actually recorded from your program.*

2. Write the code to *compute* the shadow map (but not render shadows using it), including computing all of the input that it needs. It is OK if this doesn't run, but it should be a few typos away from correct, not missing whole variables.

2.2 Specification Exercise **(due Monday 12:00 pm in SVN and hardcopy)**

Work with your *project 7* (i.e., next week's) group to write a specification for project 7, as described below, and commit it as `specification.dox` in the root directory of your *project 7* directory. Note that you can schedule time with your future partners but should not actually begin work on the implementation before Tuesday.

This Thursday at the start of lab, I'm going to give you a specification for *next* week's project and enable your SVN directory for that project . You won't implement that specification this week. Instead, you'll have until Monday to work with your project 7 group to revise that specification and submit it. The following week, you'll implement your own revised specification. You can renegotiate it with me during the week as well, but I'm more amenable to changes before work starts.

The goal of this exercise is to practice scoping work in preparation for creating your final project specification. I'll evaluate your project 7 work based on your own specification, taking into consideration both how much you accomplished and how well you accomplished it.

Your task *this* week is to select the features that will give you the **best educational value** and **presentation potential** for your limited development time. Avoid the point of diminishing returns, where you need a lot more time to produce minimal value. Focus on elements that give either proportional value for your time, or better yet, where a small amount of work gives a large payoff.

Choose your own partner(s) for project 7. You may work in a group of 1-4 people. As always, I take the number of people into account when evaluating your work.

3 Implementation Advice

3.1 Modeling

Begin with the (latest) default G3D starter project, which iCompile will produce for you. Turn down the rendering rate to something reasonable, like `setDesiredFrameRate(60)`.

Modify your scene to contain only point lights and to load a Quake 3 level.

Choose a Quake 3 scene for which there are large areas open to the sky so that you can initially have a single source that is the sun. You can use any of the levels provided in G3D or download your own from your favorite mod (there are hundreds online!) Note that you can use the G3D viewer program to preview these without writing any code. (It doesn't use `ArticulatedModel` to load them, so you can't follow its source, however.)

The textures will be missing for many Quake maps. You can see the names of the textures that are missing in your `log.txt` file. If you create textures with the appropriate names you can trick G3D into loading them instead of leaving those areas white. You can also grab one of the many open source Quake texture packs and put it on your computer to avoid the problem.

3.2 Shading

Replace `App::onGraphics3D` with your own code to explicitly render the indexed triangle lists stored within the `UniversalSurface::GPUGeoms` of the `surface` array. You will have to downcast each `shared_ptr<Surface>` to a `shared_ptr<UniversalSurface>` to access its geometry.

Create a direct illumination shader. Initially have it just set the color of each pixel based on the normal so that you can debug your transformations:

Tip: See the "GPU Model" tab at the top of the documentation.

```
#version 120 // -*- c++ -*-
/** \file direct.vrt*/

attribute vec3 positionIn;
attribute vec3 normalIn;
attribute vec2 texCoordIn;

...

/** World space normal */
varying vec3 n;

/** World space point being shaded */
varying vec3 X;

void main() {
    X = ???;
    n = ???;

    gl_Position = ???;
}
```

```
#version 120 // -*- c++ -*-
/** \file direct.pix */

/** Interpolated world space normal [not unit length] */
varying vec3 n;

/** World space point being shaded */
varying vec3 X;

void main() {
    n = normalize(n);
    gl_FragData[0].rgb = n * 0.5 + vec3(0.5);
}
```

You must complete the "`???`" sections yourself following the G3D documentation and your notes from last week's tutorial.

Once you know the transformation is correct, also pass the texture coordinate from the vertex shader to the pixel shader.

Now pass the Lambertian BSDF material property of each `GPUGeom` to the shader. The constant is a `vec4` in GLSL (red, green, blue, and alpha-coverage). The texture has type `sampler2D` in GLSL. Because it can be `NULL` in C++, use `Texture::whiteIfNull` to ensure that you never pass a `NULL` pointer to the GPU. For now, skip the glossy and other terms. You can optionally implement them later if you like (see `data/UniversalSurface_render.pix` to see the unpacking algorithm for the glossy part–it is somewhat complicated.) Render each surface using its Lambertian component (the product of the texture map sample and the constant) as a debugging step.

Tip: You're going to have to refer to the GLSL documentation at this stage, and will have to figure out what to do with the alpha channels. The Graphics Codex also contains a list of all GLSL functions.

Pass the world-space position of the light to your shader. Use this to compute the direct illumination at each point. The code should look exactly like your ray tracer, except using GLSL syntax instead of G3D/C++ syntax. Your GLSL code is going to look almost the same as your code from the early versions of the ray tracer.

Remember to check your units–the final output from the pixel shader should have radiance units.

Tip: It is helpful to temporarily disable the Lambertian texture by setting it to white so that you can really see your lighting.

3.3 Environment Lighting

You'll notice that the back sides of objects are very dark. That's because we have no model of indirect lighting. The "environmentMap" cube map is used to approximate indirect lighting. It is a picture of that ideally represents what you would see if you stood in the center of the scene. That is, it is $L_{\text{in}}(X, \omega_{\text{i}})$ for some fixed point X.

To get true indirect lighting from an environment map, we'd need a different and correct environment map for every single point in the scene. We'd also have to consider incident light from all possible directions (these cube maps tend to be at least 512×512 pixels for each of 6 faces, so that's a lot of directions to handle!)

We can use a really cheap approximation, however. The lowest MIP level is an average of a large number of directions. If we assume that the cube map is a constant, we can directly integrate over the hemisphere. If we assume that it is piecewise constant, we can perform a really coarse approximation by just considering the six directions that represent the faces.

My C++ code to pass the environment map to my shader is:

```
args.setUniform("environmentConstant",
    m_scene->lighting()->environmentMapConstant);

args.setUniform("environmentMap",
    Texture::whiteCubeIfNull(m_scene->lighting()->environmentMapTexture));
```

The environmentMap has GLSL type `samplerCube` and is typically read using `textureCube(sampler, direction)`. We want to force a low MIP map, so use `textureCubeLod(sampler, direction, 9.0)` for a 512×512 texture. My sampling code looks like:

```
vec3 E_ambient = environmentConstant *
 (max(0.0,  n.y) * textureCubeLod(environmentMap, vec3( 0.0,  1.0,  0.0), 9.0).rgb +
  max(0.0, -n.y) * textureCubeLod(environmentMap, vec3( 0.0, -1.0,  0.0), 9.0).rgb +
   ...);
```

3.4 Creating a Shadow Map

The G3D Scene includes a Lighting member, and that has a lightArray field of Lights...and *those* have shadow map members. If you ensure that your scene has a single shadow-casting light, then you can extract the ShadowMap from it and never create one on your own. Recall from Friday's sample midterm presentation that a shadow map is the depth buffer from a camera placed at the light. The G3D shadow map class abstracts the rendering of that depth buffer–it looks essentially the same as the code you've already written, and you can examine the G3D source code to see how it does so very efficiently. You don't have to write much code to *create* the shadow map's depth map each frame–it is mostly abstracted in G3D for you because it isn't very interesting. However, you need to use that depth map to create the shadows in your direct illumination from the viewer's point of view. That's the interesting part, and it will appear in your pixel shader.

Tip: I recommend that you search through the G3D sources as well as reading the documentation for examples of how to create a shadow map; it is intended that reading the source code is part of how people learn to use the library.

The first step is visualizing the shadow map so that you can debug. Create a `GuiTextureBox` for viewing the `ShadowMap::depthTexture()`, as shown in Figure 2. The shadow map is created for you as a field of each shadow-casting `Light` specified in the .scn.any file. Note that you can set the texture used by a `GuiTextureBox` after it is created–so you might create the texture box inside `App::makeGUI`, but not actually set it to display anything until `App::loadScene`, when the lights are created for you by the scene loader.

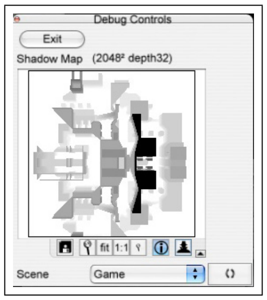

Figure 2: *Debugging GUI with shadow map visualization.*

Every frame, compute the shadow map from your scene (you obviously must do this before you attempt to shade the scene!) `ShadowMap::updateDepth` requires several arguments describing the scene and the orientation of the virtual camera placed at the light source. The `ShadowMap` class also contains static helper methods for computing these. You can see an example of their use in `Surface.cpp`.

Tip: Also look at the `Surface` static methods.

When you update the shadow map depth, reduce the bias to a small number, like $0.0f$, and specify `RenderDevice::CULL_FRONT` for the cull face. The latter inverts the normal backface removal: front faces will be ignored and back faces will be drawn.

3.5 Using a Shadow Map

When you have computed a shadow map that looks reasonable, pass the shadow map and the light's (biased) view and projection matrix to the shader using something like:

```
args.setUniform("shadowMap",
     light->shadowMap()->depthTexture());

args.setUniform("lightBiasedModelViewProjectionMatrix",
    light->shadowMap()->biasedLightMVP());
```

In your direct illumination shader, the shadow map has GLSL type `sampler2DShadow`.

The GLSL function `shadow2D(sampler, P)` returns a color as a `vec4`, but only the first element (`r`) is useful. The interpretation is that it is 1.0 if the point is visible to the light (i.e., lit) and 0.0 if the point is not visible (i.e., shadowed). It is a value between 1.0 and 0.0 if the point is partly shadowed. So you can just scale the light's power by this value at each pixel to create shadows.

The `shadow2D` call computes its result under specific assumptions about the `P` argument. It assumes that texture coordinate `(P.x, P.y)` in the shadow map that is the projection of the point being shaded under the virtual camera that we previously placed at the light source. It assumes that `P.z` represents the same point's distance along the light's "view vector" as encoded in a funny way by the projection matrix. That funny encoding is described in the OpenGL manual under `gluPerspective` and in your textbook. The basic idea is that `P.z` is on the range [0, 1], where 0 means "close to the light" and 1 means "far from the light", and the scaling is hyperbolic between them.

The key idea is that we don't care about the particular scaling of `P.z`...we just care that the same scaling previously happened when rendering the shadow map. So if `P.z` is greater than the value in the shadow map, then our world-space point `X` is farther from the light than some other surface and is in shadow. Otherwise it is the first surface seen by the light and is lit. `shadow2D` performs that comparison for us, so we just need the result.

But how do you compute `P.z`? It is the projection of `X` into the virtual light-camera's screen space...*including the homogeneous division* so that `P.w = 1`. (You don't need to pass a `vec4` to `shadow2D`; I'm just making the point because it is important to both the math and the computation you go through to produce `P`.) This is a complicated way of leading you to write two lines of code, but those two lines require a lot of thought and will probably require you to go back and read up on projection matrices again.

Finally, performing this projection in the pixel shader means that we're performing a matrix product at every pixel. The 4×4 matrix product (but not the division) can actually be lifted up to the vertex shader. You can optimally make that transformation to your code. Note the optional specification question asking why this transformation preserves correctness.

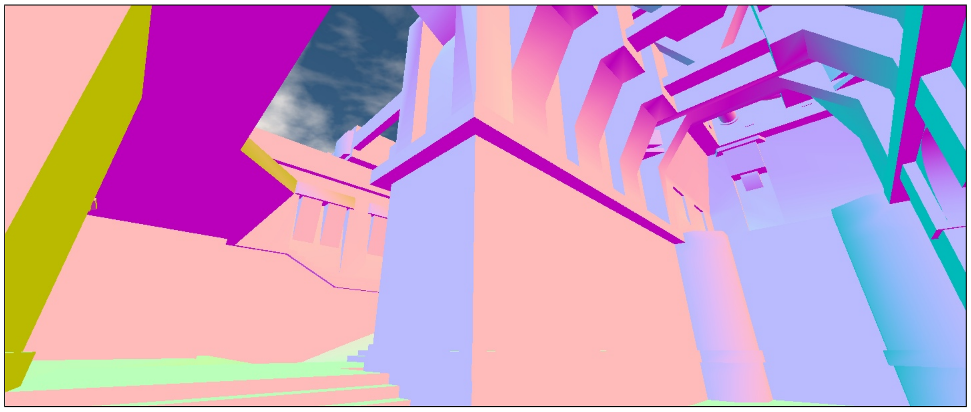

Figure 3: *Visualization of surface normals.*

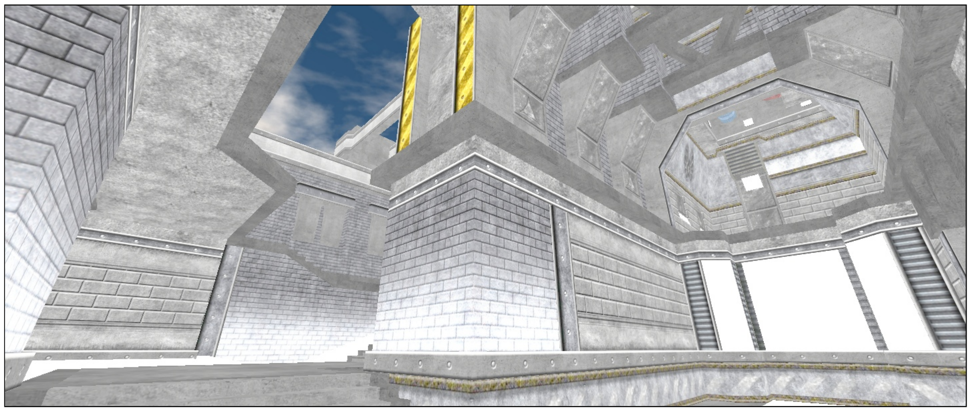

Figure 4: *Visualization of lambertian BSDF coefficients.*

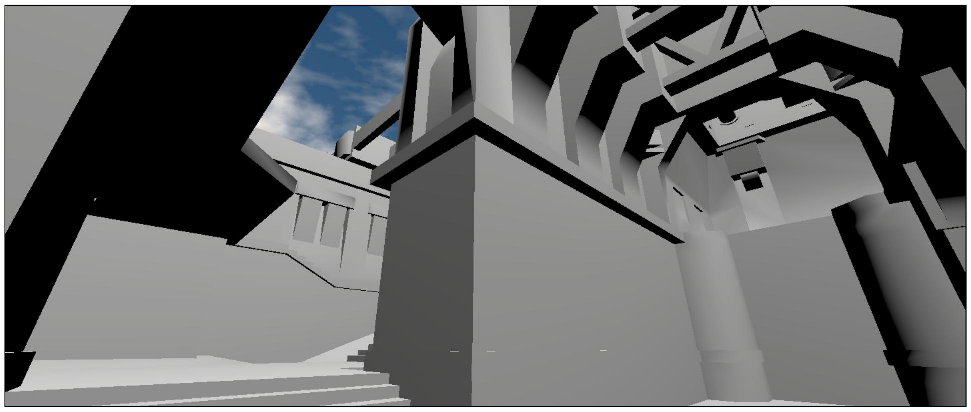

Figure 5: *Visualization of shading with BSDF disabled.*

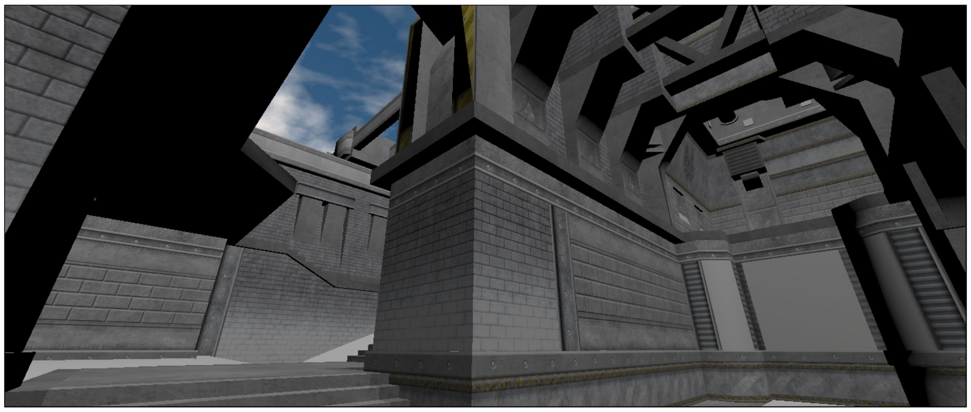

Figure 6: *Lighting and texture combined.*

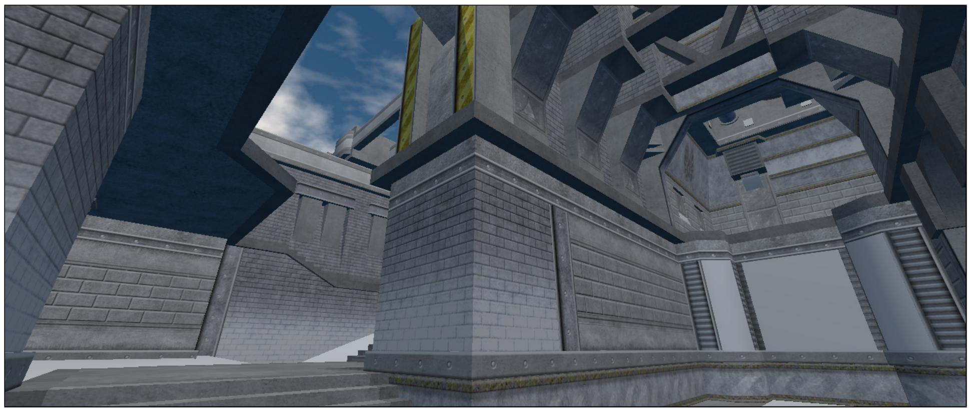

Figure 7: *Environment ("ambient") lighting approximated from a cube map.*

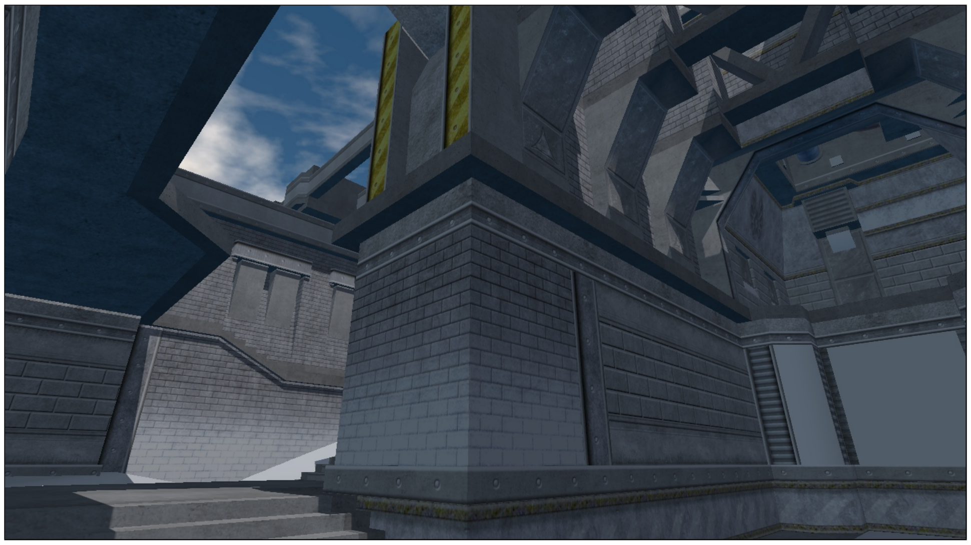

Figure 8: *Lighting with shadows. Note that the ambient light colors the shadowed regions.*

References

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph. 12*, 3, 270–274.
2