CS 371 Project 2:

Eye Rays

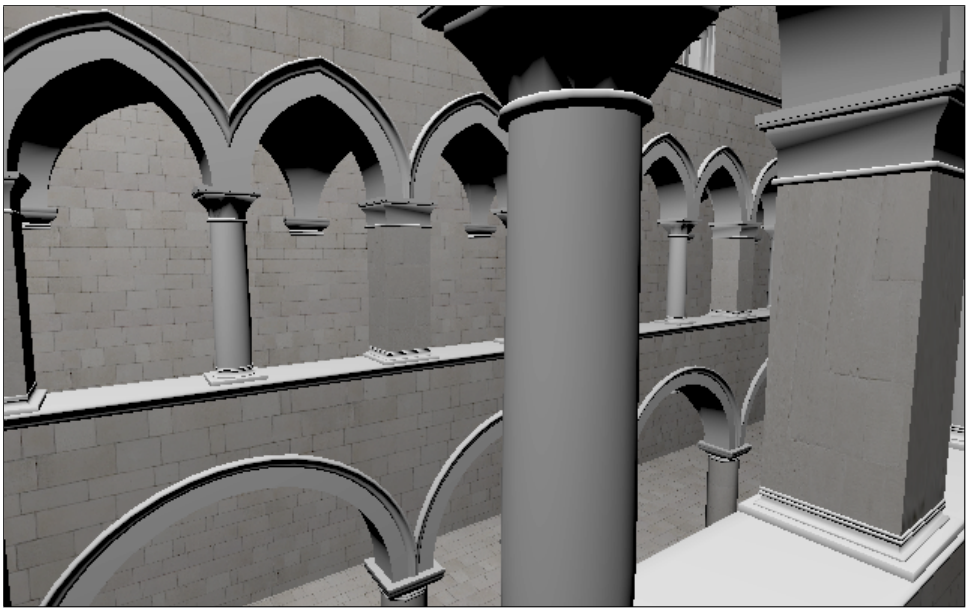

Figure 1: *Marco Dabrovic's model of the atrium of the Sponza palace (http://hdri.cgtechniques.com/~sponza/) has become a standard benchmark scene for 3D rendering algorithms. By the end of this lab, you'll be able to produce images of it yourself from first principles, without relying on the OpenGL/G3D renderer.*

1 Introduction

1.1 Overview

Ray casting is one of the core techniques for approximating photorealistic rendering. This algorithm casts a ray through each pixel to find the surface that colors that pixel. The most straightforward variant, which you will implement in this project, then shades the surface by iterating over the light sources. It was first investigated by Appel and others in the late 1960's, and quickly evolved into Whitted's **ray tracing** algorithm. Ray casting was also the basis for most real-time rendering until fairly recently. Many 3D games in the early 1990's such as *Wolfenstein*, *Doom*, *Heretic*, *Duke Nukem 3D*, and *Star Wars: Dark Forces* explicitly cast rays. Other games used a variant on ray casting called **rasterization** with **direct illumination** well into the 2000's. Later projects in this class will explore both ray tracing and rasterization using the physically-based framework that you build this week.

The previous two projects focused on modeling scenes and left the rendering to a simple black-box renderer built on OpenGL. In this project, you'll augment that preview with your own renderer so that you understand the entire system, from the data files that describe the scene to the way the value of an individual pixel is calculated. The images that your program generates this week should exactly match the ones produced by the preview renderer.

Contents

1.2 Educational Goals

On this project you'll learn about:

1. "Per-pixel" graphics
2. Ray casting
3. Direct illumination
   (radiance from the source to the surface that scatters it towards the eye)
4. The functor design pattern, applied to ray-triangle intersection
5. A first experience with concurrent graphics programming

1.3 Schedule

Out:	Tuesday,	September 21	
Checkpoint:	Thursday,	September 23,	1:00 pm
Due:	Monday,	September 27,	10:00 pm

This is an easy, solo project. As with other projects, try to quickly cover the entire specification with stubbed out methods and provisional report text before refining any one area.

As a reference, my solution required about 300 statements and 300 comment lines, including the report Doxygen comments (as reported by iCompile), plus several data files. Note that I always line-wrap my comments with M-q; if you don't your comments may look ugly in my editor and they will report as many fewer lines. If your codebase looks like it is going to be more than 1.5× larger or smaller than my solution, come talk to me because you may be on a bad path.

2 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. You may look at and use anyone's code from *last week's* project, with their permission.

You must create the Sponza scene on your own, including camera placement. You may directly share data for all other scenes, and can discuss the location of files needed for the Sponza scene openly.

During this project, you are not permitted to directly invoke the following classes and methods or look at their source code: the G3D `rayTrace` sample program and the `SuperBSDF` class.

3 Specification

Implement the following by extending the Meshes project from last week with:

Although we're implementing ray *casting* this week, next week we'll be ray *tracing*, which is why we are naming the class `RayTracer`.

1. A class named `RenderSettings` that has at least the implementation described in Section 3.3.
2. A class named `RayTracer` with at least the methods described in Section 3.4 that:
   (a) Renders images by the ray casting algorithm (casting one eye ray per pixel into the scene and computing the light scattered towards the camera from light sources)
   (b) Can operate in both single-threaded and concurrent multi-threaded modes.
   (c) Searches for intersections using both exhaustive array and pruning tree (`G3D::TriTree`) search, and can switch between them at run time.
   (d) Computes Lambertian shading under point lights as described in Equations 1 and 2.
3. A scene containing the Sponza atrium that *approximately* matches the one from Figure 1, in which:
   (a) the Sponza was loaded with default `ArticulatedModel` parameters; it is not scaled, is centered at the origin, and has its default materials
   (b) The light source was created from:

```
GLight {
    position = Vector4(15, 30, -5.5, 1.0),
    spotTarget = Vector3(-7,0,0),
    spotHalfAngle = 0.6,
    color = Power3(10000)
}
```

4. A GUI comprising:
   (a) Real-time preview of the scene.
   (b) A drop-down list for selecting a scene, with a "reload" button.
   (c) A drop-down list for selecting the ray casting resolution.
   (d) A button labeled "render" to launch the ray casting algorithm.*
   (e) Display of the last ray-cast result labeled with the wall-clock time to render it.*
   (f) Display of the number of triangles in the scene.*
   (g) Controls for enabling multi-threaded rendering.*
   (h) Controls for enabling use of the `TriTree`.*

   * New this week.
5. Analysis of the runtime of the program as described in Section 3.2
6. Create the documentation reports specified in Sections 3.1 and 3.2.

3.1 Checkpoint (Thursday 1:00 pm)

For this checkpoint:

1. Create the entire report with placeholder text and images.
2. Commit the files for the `RayTracer` and `RenderSettings` classes.
3. Implement, using a combination of pseudo-code comments and actual code, the entire `RayTracer` class. It should compile without errors but need not actually do anything or have a GUI.

3.2 Report

Write an appropriately-formatted report that covers the following topics:

1. An architectural overview of your program.
2. Discuss significant design choices that you made, and argue why your choices were good for this project.
3. Discuss any known errors in your program, and how you identified and attempted to correct them.
4. Show pictures of the following scenes rendered with ray casting:
   (a) The Cornell Box scene
   (b) The Sponza scene
   (c) A visually compelling scene of your choice that demonstrates texture mapping. This need not be a *new* scene; you can use a scene that you or another student created for a previous project, or extend such a scene. Remember to commit the scene file and anything not in `cs-local` that it needs to run to the `data-files` directory.
5. **Questions.** (*To calibrate your level of effort, all of these together should take you more than 10 minutes and less than one hour to complete.*)
   (a) The ray casting algorithm assumed that the only significant incident light was directly from the sources. Describe the errors contributed by this approach, a scene for which this error is significant, and briefly propose an algorithm for incorporating indirect light that has scattered from other surfaces.
   (b) Without performing a formal experiment, suggest the performance impact of multithreading. Is the speedup proportional to the number of cores in the computer?
   (c) Briefly speculate on how `TriTree` might work. You may research this or read the source code, but I'm more interested in your own ideas about how you would design a data structure for ray-triangle intersection.

Tip: Your ray-cast images should match those from preview mode without wireframes when `environmentMapConstant` is zero, except for pixels colored by the sky box.

(d) Briefly describe the different ways that you would have to change your program to incorporate another type of primitive, such as a true sphere. Consider everything from the scene data files through the shading algorithms.

6. **Feedback.** Your feedback is important to me for tuning the upcoming projects and lectures. Please report:

   (a) How many hours you spent **outside** of class on this project on **required** elements, i.e., the minimum needed to satisfy the specification.

   (b) How many additional hours you spent outside of class on this project on **optional** elements, such as polishing your custom scene or extreme formatting of the report.

   (c) Rate the difficulty of this project for this point in a 300-level course as: too easy, easy, moderate, challenging, or too hard. What made it so?

   (d) What did you learn on this project (very briefly)? In addition to the algorithm, consider the workflow lessons, programming and design experience, and the process of thinking about questions. Rate the educational value relative to the time invested from 1 (low) to 5 (high).

3.3 `RenderSettings`

The `RenderSettings` class describes all of the options related to image formation that are not part of the camera (as we have abstracted it) or scene. You may extend this, but for to ensure compatibility between each other's projects, please implement at least the following interface. You may copy-paste the header file, but please re-type the C++ file to ensure that you read it closely. As with all code in your program, you are responsible for understanding, documenting, and debugging this code.

```
class RenderSettings {
protected:

    GuiDropDownList* m_resolutionList;

public:

    /** Image width in pixels */
    int               width;

    /** Image height in pixels */
    int               height;

    bool              multithreaded;
    bool              useTriTree;

    RenderSettings();
    virtual ~RenderSettings() {}

    /** GUI callback */
```

```
    void onResolutionChange();

    /** Called from App::makeGui */
    void makeGui(GuiPane* p);
};
```

The implementation of the required methods is:

```
// You can change these defaults:
RenderSettings::RenderSettings() :
    width(256), height(128), multithreaded(true),
    useTriTree(false) {}


void RenderSettings::onResolutionChange() {
    TextInput ti(TextInput::FROM_STRING,
         m_resolutionList->selectedValue().text());

    width = ti.readNumber();
    ti.readSymbol("x");
    height = ti.readNumber();
}


void RenderSettings::makeGui(GuiPane* p) {
    p->setNewChildSize(250, GuiPane::DEFAULT_SIZE, 120);

    static const Array<std::string> resArray
        ("256 x 128",
         "320 x 200",
         "640 x 400",
         "1280 x 720");

    m_resolutionList = p->addDropDownList
        ("Resolution", resArray, NULL,
         GuiControl::Callback(this,
            &RenderSettings::onResolutionChange));

    // Add more GUI controls here...
}
```

Tip: I'm giving you code as source here and through the G3D library, but not telling you how it is intended to be used. Discuss this with your classmates who may have different insights.

3.4 `RayTracer`

There are many ways of structuring a `RayTracer` class. I'm asking you to extend the following structure because it will reduce the amount of refactoring you need on subsequent assignments. You may copy and paste this interface into a header file and need not re-type it.

```
protected:
    /** Trace all pixels in this region and write the
        results to m_image.  Not threadsafe. */
    void backwardTrace(int x0, int y0, int x1, int y1);
```

```
    /** Trace this pixel only and write the result to
        m_image. Threadsafe. */
    void backwardTrace(int x, int y);

    /** Estimates \f$L_{\mathrm{i}}(X, \hat{\omega}_{\mathrm{i}})\f$
        for the ray, where \f$\hat{\omega}_{\mathrm{i}}\f$ =
        <code>-ray.direction()</code>.

        \param backwardBouncesLeft Reserved for future use.
        */
    Radiance3 backwardTrace
        (const Ray& ray,
         int backwardBouncesLeft) const;

public:

    virtual ~RayTracer() {}

    enum {ALL = -1};

    /** Returns the number of triangles in the scene */
    int setScene(const Scene::Ref& scene);

    /** Blocks until the entire image is rendered.

        Not threadsafe. Only invoke this method on a
        single thread per RayTracer instance.

        \return An image containing an estimate of the
         radiance through each pixel center.

        \param pixel The pixel to trace (for debugging
           purposes)
    */
    Image3::Ref render
    (const RenderSettings& settings,
     const GCamera&        camera,
     Vector2int16          pixel = Vector2int16(ALL, ALL));
```

4 Evaluation Metrics

Recall that you are responsible for the entire project, including the correctness and clarity of code that you exported from Subversion to start the project, even if you were not the original author. So make sure that you know what code is in your project and that you understand it all!

To evaluate your project, I will check your project out from Subversion as of the deadline time. I will then run `icompile --doc` to generate the final report and documentation. I will read sections of your source code, the report in the `index.html` page generated by Doxygen, and sections of your documentation as generated by Doxygen. I may run your program, but I will primarily investigate its functionality by the description that you provide in the report. Note under this scheme, that the artifacts from your creation of and experimentation with the program are more important than the executable program itself. For many projects you can receive a favorable evaluation even if your program does not compile or execute.

As described in the *Welcome to Computer Graphics* document, I will evaluate your project in several categories:

- Mathematical (algorithm, geometry, physics) correctness
- Adherence to the specification
- Program quality
- Report quality

Some questions I consider when evaluating the source code are: Is it possible for someone unfamiliar with it to find specific routines quickly? Is the code easy to understand? Does it make good tradeoffs between efficiency, clarity, and flexibility? Are data structures used effectively? Are the algorithms correct? Are the geometry and physics correct?

When evaluating the report, I consider: Do the experiments adequately explore the correctness, performance, robustness, and parameter space of the algorithm? Are known bugs made clear, along with how you tried to solve them? Are appropriate sources cited for algorithms and code? Does the overview documentation guide a reader to the relevant source code documentation? Is the architecture of the program clear?

The report and code should both be as concise as possible without compromising clarity.

Tip: You need the `TriTree` and multithreading to be able to render large scenes in a reasonable amount of time. If you are unable to get these working, do not panic, and do not wait for hours for Sponza to render using `Array`. As long as you have written the code to attempt to use `TriTree` and multithreading and have documented the problem and how you tried to overcome it, you lose few points for the minor issue of not having a final image of Sponza in your report.

5 Implementation Advice

5.1 Getting Started

Start by exporting the previous week's code to new Subversion project. See the Tools handout from last week or refer to the Subversion manual [**?**] for information about how to do this. Remember that you can use *anybody's* code from the previous week with their permission, so if you aren't happy with your own project as a starting point, just ask around.

The Subversion command to check out your project this week is:

```
svn co svn://graphics-svn.cs.williams.edu/2-EyeRays/eyerays-<name>
```

in which you should replace `<name>` with your user name.

5.2 The Rendering GUI

Your `RayTracer::render` method will take a relatively long time to compute an image, perhaps several minutes. If you put it in `App::onGraphics`, it will run every time that the screen needs to refresh. That will make it appear that your program has crashed. So you should only invoke `render` when the user presses the "Render" button. Look at the starter code for the exit and scene-reload buttons to see how to connect a button to a method of `App`.

You can time infrequent tasks by measuring the difference in `G3D::System::time` calls. For tasks that run every frame the `G3D::Profiler` and `G3D::StopWatch` classes are more appropriate, but you don't need that on this project.

There are many ways to display your image on screen. You could convert the CPU-image that you rendered into a GPU-image using one of the many `G3D::Texture` constructors, and then write code in `App::onGraphics2D` that draws a rectangle filled with that texture. For this approach, you may wish to use `G3D::RenderDevice::push2D` and `G3D::Draw::rect2D`.

Alternatively, you can use the `G3D::GApp::show` method to create a pop-up window with your image inside it (that's what I did, since it was really easy; see Figure 2). That method just creates a new `G3D::GuiWindow` with a single `G3D::GuiTextureBox` inside it, so you could also create your own style of pop-up window, or embed the display within another part of your UI.

The drawback of using the built-in G3D GUI controls to display your image is that it is hard to add your own debugging handlers. For example, if you explicitly render your own UI in `onGraphics2D`, then you can write an `onEvent` handler that detects mouse clicks on them. It is often handy when debugging a ray caster to launch a 1×1 window, i.e., single-ray, render job when the user clicks on a pixel. This allows you to render the whole scene, and then set a breakpoint and re-cast the ray through one pixel while watching it in the debugger. It is more challenging to set up that kind of infrastructure if you are using a GUI control.

There are many ways to display the number of triangles in the scene. For example, you can print on-screen using a `G3D::screenPrintf` from `onGraphics3D`, an explicit call to `G3D::GFont::draw2D` from `onGraphics2D`, create a disabled `G3D::GuiNumberBox` or `G3D::GuiTextBox`, or create a `G3D::GuiLabel`. As with your other UI choices, you must decide how much you value ease of imple-

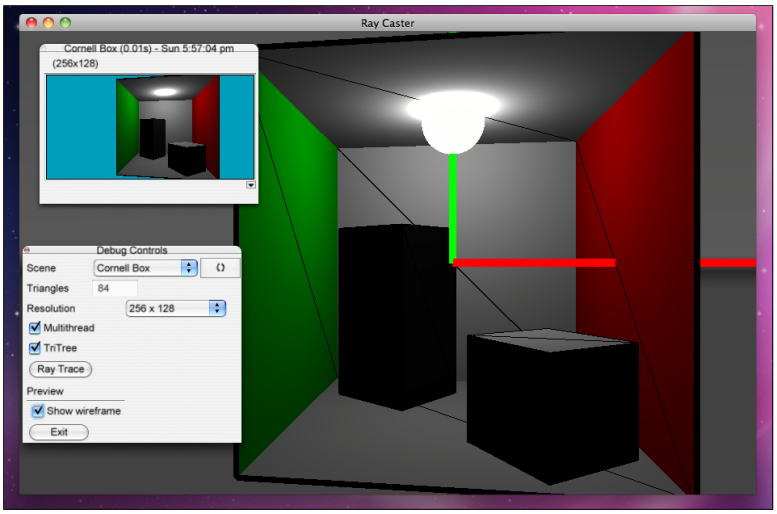

Figure 2: *One possible user interface for the ray casting program.*

mentation, ease of use for the end-user, attractiveness, performance, and functionality. Mine looked like:

```
debugPane->addNumberBox("Triangles", &m_triangleCount, "")->setEnabled(false);
```

Remember to briefly describe and support your UI choices in the report, just as you would for your other implementation decisions.

5.3 `TriTree`

The `TriTree` data structure is a binary tree of triangles used to accelerate the ray casting algorithm. We'll study how it works later in the semester, and next week we'll perform detailed experiments to determine exactly how much faster it makes the program.

When `RayTracer::setScene` is invoked, it should convert the `Scene` into a form that allows both array and `TriTree` iteration. There are several ways to do this. I found the easiest to be posing the scene and then calling `G3D::TriTree::setContents` on the `Surface::Ref` array[1]. I never created an explicit `Array<Tri>`; instead, I used the `G3D::TriTree::operator[]` and `size` methods to treat the `TriTree` as if it were an array. You can follow this design or create an alternate one.

When G3D loads a scene, it assumes that you are going perform hardware rendering on the graphics card (GPU), which has a separate memory space from the CPU. Before you can invoke the CPU-methods on the `G3D::Material`, you must therefore copy the texture data to the CPU. The `G3D::Material` methods actually do this for you automatically, but are not threadsafe, so when writing a multi-threaded program you have to explicitly perform the data copy on the main thread before launching additional threads. The `G3D::TriTree::setContents` method takes an optional `G3D::ImageStorage` argument. Pass `COPY_TO_CPU` to tell it to copy texture data from the GPU to the CPU while it is extracting the triangles from the scene.

[1] `SurfaceRef` is an alias for `Surface::Ref` needed to break a dependency in the API.

5.4 Functors (Closures)

The **functor** design pattern is a C++ class that acts like a C++ function. They are an approximation of the general programming language feature of a **closure**, which is just a function that has a persistent parent environment in which to retain state.

A regular C++ function can retain state between invocations. Local variables marked with the `static` keyword are initialized once, the first time that the function is invoked, and then retain their value on subsequent evaluations. This is convenient for memoizing results, for example. However, it has several drawbacks. One drawback is that the programmer has little control over the order in which variables are destroyed when the program shuts down. Another drawback is that it is hard for other parts of the program to access the state stored in these local variables.

C++ allows overloading of several operators. For example, we can write

```
Vector3 a;
Vector3 b;
...
a = a + b;
```

because `Vector3::operator+` overloads the default `+` operator, which is only defined for numbers. In addition to the expected arithmetic operators, some surprising operators can be overloaded. These include the dereference operator, `->`, the array operator, `[]`, and the **function application operator**, `()`. This means that we can create a class that syntactically acts like a function. For example:

```
class fakeFunction {
public:
     float operator()(int x, bool y);
};

...

float z = fakeFunction(3, true);
```

Such a class is called a **functor**. Because it is actually a class, we can add member variables and other utility methods. That allows the "function" to retain state between invocations, and for access to that state from other parts of the program.

The `G3D::Tri::Intersector` class that you will use in the ray caster is a functor. It exposes the triangle and the barycentric coordinates of the closest intersection as public member variables. It provides additional information about the intersection through some helper methods. It also has some member variables that we won't use in this project that control how the `operator()` works.

5.5 Ray Casting Algorithm

The **ray casting** algorithm that you are implementing is:

For each pixel position (x, y):

1. **Generate** the ray R from the camera aperture through the center of the pixel position (x, y).
2. Find the first (i.e., closest to the aperture) **intersection** H of ray R with the scene triangles
3. If there is no intersection then
   (a) Set the pixel to a constant "**background**" color

   else
   (a) Set the pixel's value to the **shade** of the surface.

(The boldfaced words are intended to suggest some variable and methods that you may wish to use or implement.)

5.6 Iterating Over Pixels

For single-threaded iteration, I assume that you can figure out how to write the necessary loops to iterate over pixels. Perform your iteration in a cache-friendly fashion. To do so you'll have to discover how your image is actually stored in memory. When debugging, always run in single-threaded mode. It is much harder to debug a multi-threaded program.

For multi-threaded iteration, you have two choices. You can either create a `ThreadSet` of `GThread` subclasses manually, or invoke `GThread::runConcurrently2D` with a callback method. I chose the latter in my own implementation.

5.7 Generating Rays

For a camera at the origin facing along the $-z$-axis, it is a simple matter of geometry to compute the ray from the eye. For a camera with an arbitrary orientation that computation becomes more complex than we are prepared to deal with at the moment. So invoke `G3D::GCamera::worldRay` to generate the ray through a pixel. Note that `worldRay` considers (0, 0) the upper-left corner of the upper-left pixel, so (0.5, 0.5) is the *center* of the upper-left pixel. Refer to the *CS371 Tools* overview and G3D documentation for the 2D coordinate system specification.

Tip: You `App` needs a `RayTracer` member. Consider the merits of making this member of type `RayTracer*` vs. `RayTracer::Ref` vs. `RayTracer`.

5.8 Finding Intersection

You can compute the ray-triangle intersection yourself, use the helper methods on `G3D::Ray`, or use the `G3D::Tri::Intersector` class.

Once you have found the intersection, there are several ways of extracting the information that you need for shading. I chose to create a `G3D::SurfaceSample` from the `G3D::Tri::Intersector` that I used to find the intersection, however you are welcome to explore other designs.

5.9 Shading

Shade surface points visible to the eye using the equation:

$$\text{Let } \hat{\omega}_\text{i} = \mathbf{S}^2(Q_j - X)$$
$$\text{Let } d = ||Q_j - X||$$
$$L_\text{o}(X, \hat{\omega}_\text{o}) = \sum_{j=0}^{N-1} \left[\frac{\Phi_j}{4\pi d^2} \left(\hat{\omega}_\text{i} \cdot \hat{n}\right) f_X(\hat{\omega}_\text{i}, \hat{\omega}_\text{o}, \hat{n}) \right] \quad (1)$$

where light with index $0 \leq j < N$ is described by power Φ_j and position Q_j, X is the intersection point, $\hat{n}$ is the **shading normal** at the intersection, $\hat{\omega}_\text{o}$ points back along the ray.

For this week, we'll assume that f_X is constant with respect to the incident direction (as long as it is on the same side as the light!) and implement it as:

$$f_X(\hat{\omega}_\text{i}, \hat{\omega}_\text{o}, \hat{n}) = \begin{cases} \frac{k_X}{\pi} & \text{if } (\hat{\omega}_\text{i} \cdot \hat{n} > 0) \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

where constant k_X is given by `G3D::SurfaceSample::lambertianReflect`. Note that "0" really means `G3D::Color3::zero()` in this context; the math notation hides the dimension because shading equations are typically expressed for a single wavelength.

You should clearly document the shading function in your code and justify it in the same way that we did in class. This is a good chance to exercise your new LaTeXskills.

In the G3D documentation you will find several applicable methods and fields for implementing this method, including:

- `G3D::GLight::power`
- `G3D::GLight::position` (this gives you a `Vector4`, and you need a `Point3`; invoke the `Vector4::xyz` method for an easy conversion)
- `G3D::SurfaceSample`, which extracts all of the constants needed for shading a point on a triangle from a texture coordinate.
- `G3D::pif()`, which returns π in floating-point precision.

This week, you are not required to restrict spot lights to only illuminate points within their cone, although you may do so if you choose. Do not implement any recursive or shadow rays this week.

The `G3D::Material::bsdf` class abstracts the bidirectional scattering function f and can implement it for a large range of materials. I'm asking you to not use it this week so that you'll see all of the parts of shading. In some future projects you'll have a choice of whether to use it or to write out the shading explicitly as we did here.

Note that in mathematical notation we describe f as function parameterized on a point and three directions, but in the programming API it is parameterized on

a texture coordinate and the directions. That is because the texture map represents the varying reflectivity of the surface as a function of texture coordinate. So the `SuperBSDF::evaluate` function doesn't actually need the intersection point, only the texture coordinate to determine the applicable reflectivity.