ECE/CS 285 Advanced Image Synthesis

Homework 1

Due Monday April 30, 11:55pm

Reyes-style architecture

In this assignment, you will implement a Reyes-style, micropolygon-based architecture that processes input through a RenderMan[1]-like C interface. Although RenderMan is a sophisticated interface and writing a fully RenderMan-compatible rendering system would take a lot of time, in this assignment we will only implement a subset of the features. After all, the goal here is simply to give students get better understanding for the Reyes architecture that implements the RenderMan interface in a program such as PRMan.

1. Overview

You will implement a rendering system that takes in a scene in a RenderMan-like C language and outputs the correct image. Note that the C interface for Renderman is not really used much (most modeling programs output the scenes in the more compact RIB format), but this works for our purposes here. So we will not be using the RIB interface but rather these C-bindings instead.

For example, here is a scene that I've been using for some of my tests:

```c
void myScene(void) {
 RiBegin();

  RiFormat(100,100, 1.0);
  RiFrameAspectRatio(4.0/3.0);

  RiFrameBegin(0);

   /* set the perspective transformation */
   float fov = 45.0;
   RiProjection(RI_PERSPECTIVE, "fov", &fov);
   RiTranslate(0, 0, 5.0);


   RiWorldBegin();

    RiSphere(5.0, 5.0, 5.0, 2 * MY_PI);

   RiWorldEnd();
  RiFrameEnd();
 RiEnd();
};
```

[1] *The RenderMan® Interface Procedures are Copyright 1988, 1989, Pixar. All rights reserved. RenderMan® is a registered trademark of Pixar.*

As you can see, you will be implementing many (but thankfully not all) of the Ri*() functions pertinent to the basics of the system. Although RenderMan resources exist online, I ask you to **write your own code, and do not use available code you find on the internet or other locations**. The only exception to this is that you can use the ANSI C binding described in Appendix C.1 of the RenderMan specification as an "Ri.h" file for a starting point, if you wish. However, keep in mind that you might need to modify some of the function prototypes to get it to work with the homework assignment. You are also free to use scenes, textures, shaders etc. from other resources to test your implementation and to model your final scene as long as you document the sources properly.

At a high level, your implementation should be able to set up a rudimentary graphics state, render primitives with bound shaders, and determine the visibility using a z-buffer algorithm. Rendering primitives in a Reyes architecture involves dicing them up into a micropolygon grid, shading the grid, and then busting up the grid into micropolygons that are bounded and sampled into screen space locations. You also need to implement a "hider" to determine visibility. We will use the stochastically sampled z-buffer described in the Reyes paper by Cook et al. to do this.

2. Setting up graphics state

You will need to implement the commands that set up the basic scene and image options, such as the image format and the parameters of the camera (position, fov, projection transformation). For the projection transformation, you should be able to do both orthogographic and perspective transformations. You should also be able to save out the images into a file specified by `RiDisplay()`. You can use the CImage class from Microsoft to do this.

Other things you will have to implement to get this to work:

```
RiDisplay()
RiProjection()
```

You don't have to worry about:

- No motion blur or depth of field
- User-programmable clip planes
- Instanced objects
- RIB interfaces (assume everything is through the C binding)
- No transformation stack which would require push/pop of the transformation matrices (only one `RiTransformBegin/End()` statement at any point, no nesting)

3. Processing of primitives

In this assignment, you can assume that every primitive is "diceable," so you will not have to split any primitives. This will generate lots of micropolygons but that is ok for the assignment (performance is not a major issue, but I will frown on extremely slow implementations). You will need to bound objects as described in the specification, and to provide a dice function to break it up into micropolygons.

You will need support the following four quadric primitives:

Sphere (`RiSphere`)
Cylinder (`RiCylinder`)
Cone (`RiCone`)
Torus (`RiTorus`)

As you evaluate each point in the micropolygon grid, you will have to evaluate **P** (the position in object space of the point on the surface), the surface parameters **(u, v)**, the derivatives **(du, dv)** and **(dPdu, dPdv)**, the surface shading normal **N**. These will be available as "global variables" to the shaders.

Other things you will have to implement to get this to work:
`RiColor()` – To specify the current color of the object

You don't have to worry about: splitting, motion blur, depth of field, blobs, polygons, sub-division surfaces, NURBS, procedural geometry, objects that cross the eye plane, texture coordinates (or other variables) being specified per grid point, level of detail, etc.

4. Hider and image formation

You will need to implement a z-buffer hider similar to that described in the Reyes paper by Cook et al. The position of the samples should be in a jittered pattern for each pixel, and the number of samples should be specified by the `RiPixelSamples()` command. You should be able to apply a box filter to the sample values to reconstruct the final image (you don't need to worry about other filter types). You should output both color and opacity at each pixel, but if it is saved as a standard image then you only need to output RGB. In order to compute the color of the samples, you should linearly interpolate the corner colors of the micropolygon.

You don't have to worry about:

- No exposure modifications or quantization
- Filter can be hardcoded (box filter)
- Different number of components for color (assume always 3)
- No need to implement RiHider(), cannot be changed

5. Shading System

Unlike Renderman renderers which effectively implement a virtual machine for executing the compiled shaders, we will take short cut by writing our shaders directly in C. So the idea is that we have a separate shaders file (say shaders.cpp) and we write the shaders directly in C and call them via a **function pointer** linked from our primitive class. To do this, we will use the RiSurface() and RiDisplacement() function calls to set the geometry attribute to the shader to be used. This is meant to handle two kinds of shaders: surface and displacement.

To do this, you will need a way to communicate the global variables to the shader. Do this by defining a set of global variables (e.g. __my_shader_P, __my_shader_u, __my_shader_N, etc.)

that you set from the loop that is going through each grid element and can be accessed by the shader. The surface shader should compute the output color and opacity and at the same time the new position of the surface and the new normal.

```
/* see Table 12.1 for info on these predefined vars */
RtPoint __my_shader_P;
RtColor __my_shader_Ci;

void SimpleDisplacementShader(void) {
      __my_shader_P[1] += 2.0;      // move up the y coord of the point by 2
}



void SimpleGreenShader(void) {
      // set to green
      __my_shader_Ci [0] = 0;
      __my_shader_Ci [1] = 1;
      __my_shader_Ci [2] = 0;
}
```

And we would bind the shaders in this way:

```
RiDisplacement (SimpleDisplacementShader);
RiSurface (SimpleGreenShader);
```

This would be called right before calling the primitive function so that the shader would be executed as we are processing the micropolygons of this primitive. Before we process any micropolygons, however, you would first set the global variables (__my_shader_P, etc.) based on the values coming from the mesh and then you would call the shaders once per micropolygon.

Remember that you will have to do this process twice, one for the displacement shader and once for the surface shader. Note that the displacement shader has to be executed first so that the surface normals, etc., are updated with the new displacement so that the surface shader is correct. In other words, first run the displacement shader to adjust the positions of the points, and when executing the surface shader update the surface normals based on the new positions. After each call, you would read back the values that could have been modified back to the mesh.

You will need to implement the following shaders: standard lighting, checkerboard(), and bumpy(). The first one implements the standard lighting model (Phong model) we have seen in class, the second makes a checkerboard pattern of specified dimensions, and the last one is a displacement shader that makes the surface look bumpy using Perlin noise.

You don’t have to worry about:

- No light shaders! Instead, assume that the light sources are hardcoded in the shaders file as other C functions. You can hardcode the positions of the lights within the scene there too if you wish.
- No volume shaders, image shaders,

- No shader compilation! The C compiler takes care of that!
- No virtual machine to evaluate the shaders in SIMD. You can assume that each point is shaded completely before moving to the next (more like OpenGL) and so there is no need to do instruction-by-instruction parallel execution of the shader. This precludes computing the derivatives of arbitrary variables from within the shader, but that's fine (we simply won't be able to do some of the more complex shader antialiasing we talked about in class).

6. Textures

Use the CImage class from Microsoft (or any other image-processing library you want to use and can submit working/compiling code for) to read in a texture from a file and texture map it along the surface of the objects. Use the (u,v) coordinates (which are parameterized from 0 to 1) as the (s,t) coordinates for the texture lookup.

Because we are not going to handle everything in the same way RenderMan does, we are going to write our own texture function to make things easier to do this.

```
RiMakeTexture("my_texture.bmp", 0);
```

This will load the "my_texture" texture into the zeroth slot of the texture data structure. Next, we need a way of invoking the texture from within a shader. You should then implement a texture command so that you can call it like this:

```
RtFloat __my_shader_u, __my_shader_v;

// this shader simply does a texture lookup on the texture bound at slot zero
void SimpleTextureShader(void) {
      Ci = texture(0);
     // note that as per the spec (last paragraph in Sec. 15.7.1),
     // we assume s,t are predefined
}
```

You don't have to worry about:

- Filtering textures that are accessed from within a shader with some other coordinates than u,v

7. Transparency

You will need to implement transparency so that you can render transparent objects. This involves keeping a sorted linked list at each sample in the framebuffer, where micropolygons are sorted with respect to their distance from the camera. When an opaque object is inserted into the list, you must clear out the remaining items from the rest of the list, because that opaque sample effectively blocks the rest of the samples. Construct a scene and make an image demonstrating your ability to render transparent objects.

Other things you will have to implement to get this to work:

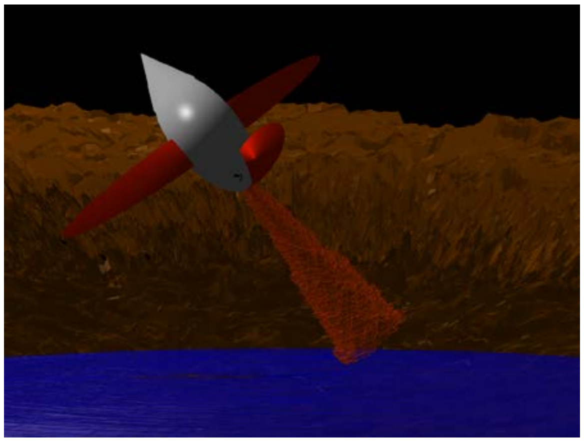
- `RiOpacity()`

8. Run your system on the test scenes I provide
To demonstrate that your Reyes system works, you will have to run it on the test scenes I provide. First, try the scene in SampleScene.cpp (which is called SampleScene1). This scene renders a moving white ball on top of a coordinate axis system. It is the first animationyou're your system! Feel free to comment out the animation steps while you are implementing stuff to make it easier to debug.

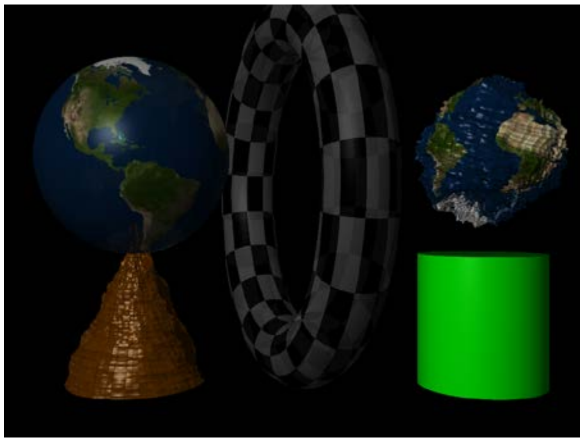
Next, there is a file called MoreScenes.cpp, which contains the following (student-designed) scenes:

Earth

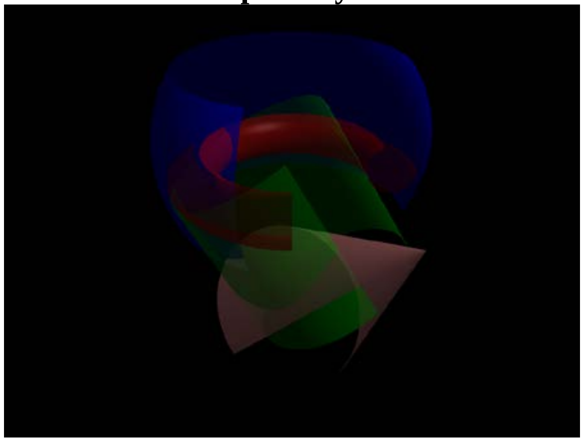
Rocket



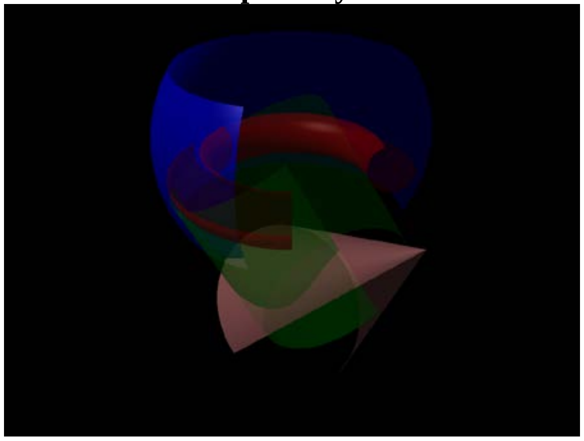
ShaderTest



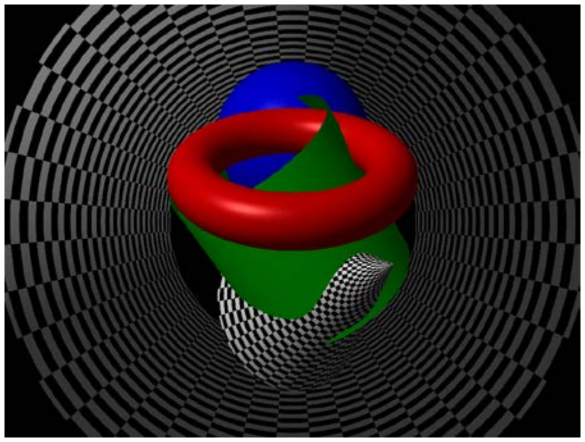
TransparencyTest1



TransparencyTest2



Tunnel



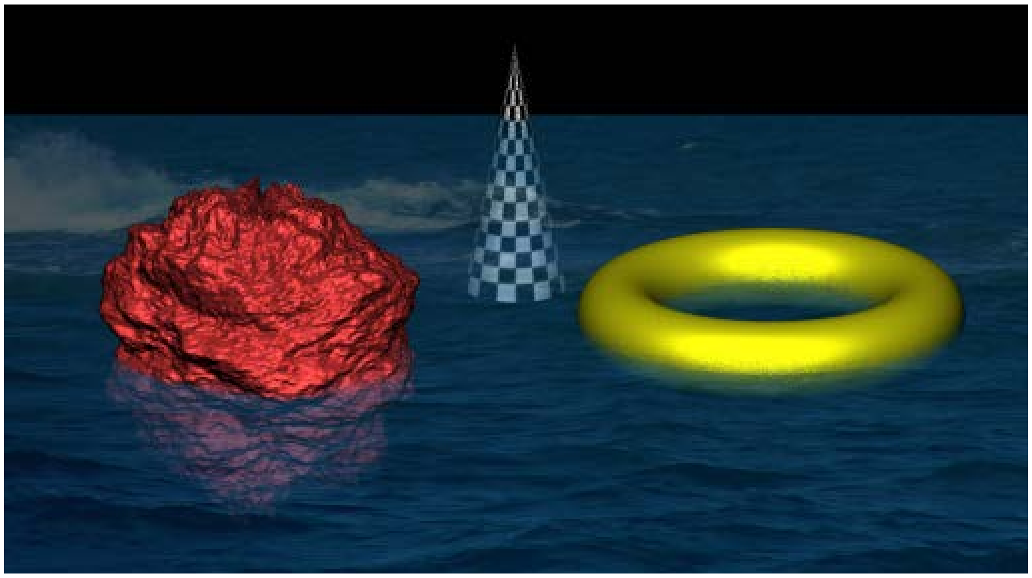
I expect your REYES implementation to produce comparable results to these scenes, so make sure you test them thoroughly. Note that you will have to write your own shaders to get these scenes to work, e.g., the checkerboard, Phong, and bumpy shaders, as described earlier in the assignment. To get things working first, I suggest that you test the scenes without the shaders first and then only add them in at the end.

9. Create your own interesting scenes

You will also need to construct several test scenes to demonstrate working behavior. Please implement the following two scenes (in addition to your normal test scenes for debugging):

- Create and render a scene that has the four main primitives (sphere, cylinder, cone, torus), but set the zmin, zmax, and thetamax for these to different values so that the shapes are incomplete and are missing their top and bottom and are open on one side (see, e.g., Fig. 5.3 in the RenderMan spec). Make them semi-transparent (part 7) as shown in this diagram so that they are easier to see.

- Create and render a test scene that shows off your different shaders (standard lighting, checkerboard, bumpy). Below is an example of one scene done by one of my students at UNM. See if you can come up with something better.



Please post these test scenes (along with your Renderman C scene code) on the class forum so that other students can see your results, use it to debug their own implementations, or point out problems with your approach. I'd expect all of the scenes to render similary using your system, although perhaps things like the bumpy shader would look differently with each version of the Perlin noise.

9. Make a nice scene

In this last part of the assignment, you need to create a nice test scene using the Renderman-like C interface. Aesthetic counts! I will probably post the results of your images in a public webpage.

10 Renderman commands that should be implemented

To get the homework assignment to work properly, in the end your rendering system should implement the following Renderman commands:

To set up graphics state

```
RiBegin
RiEnd()
RiFormat()
RiProjection()
RiFrameAspectRatio()
RiPixelSamples()
RiDisplay()
RiFrameBegin
RiFrameEnd()
RiWorldBegin()
RiWorldEnd()
```

Transformations

```
RiTransformBegin()
RiTransformEnd()
RiIdentity()
RiTransform()
RiPerspective()
RiTranslate()
RiRotate()
RiScale()
RiConcatTransform()
```

Primitives

```
RiSphere()
RiCone()
RiCylinder()
RiTorus()
```

Shading

```
RiColor()
RiOpacity()
RiMakeTexture()     (different than the “Ri.h” declaration)
RiSurface()         (different than the “Ri.h” declaration)
RiDisplacement()    (different than the “Ri.h” declaration)
```

Use this as your checklist in the end. If I am missing a function here that is needed for the final implementation, let me know!

11. Assumptions

I assumed these in my implementation so you can too! I’ve been listing assumptions in each of the individual sections, so this is by no means a complete list. It just re-iterates some of the ones I already stated.

- You can assume that there will be no errors in the Renderman scene descriptions, so you don’t have to do any error checking for the user’s input (but you should for catching your own mistakes).

- Ignore the transformation stack. Assume that the transformations are simply concatenated, and the nesting level is 1.

- You have no depth of field or motion blur.

- Your shader code does not have to execute each instruction in parallel mode (as is done by a RenderMan implementation), but you can execute the shader at each sample completely independently and using C.

- No object crosses the e-plane so you do not have to worry about splitting objects.
- I might be missing some other assumptions, so ask if you have any questions.

12. Extra Credit
This is can be a tough assignment for some, but there might be some of you that breeze through it and are looking for a bigger challenge. If you have time, here are a few extra things you can implement for more points.

- Implement Bezier bicubic patches as a new kind of primitive. Render a scene using the Utah teapot as an example.
- Implement motion blur by setting a time variable for each sample. Don't forget that you will have to compute the bounding boxes of the objects based on the time as well.
- Implement depth of field by jittering the polygon as a function of depth.
- Implement bucketing to accelerate your implementation.
- Output z-buffer information as well as color, show some examples of compositing images together by taking advantage of this extra information.
- Implement light shaders and demonstrate them working.
- Use the system to render a z-buffer from the light's perspective and demonstrate working shadows.
- Implement a complex shader like the oak wood shader we talked about in class.

Other things you are thinking about? Ask me!

13. Grading
The grading will be roughly as follows:

D or below – little or no working functionality
C – Some basic things are working but not much looks right.
B – Everything is mostly working, but perhaps one or two things have not been properly done. I expect the bulk of the students would get a B on the assignment.
A – All of the required functionality is working properly, two test scenes that meet the standard have been submitted, the final "nice scene" is very good
A+ – Everything done for an A along with some extra credit

14. Tips and advice

- Get started soon – don't wait until the last minute: you won't finish!

- To ensure that *everyone* is able to properly complete the assignment and get an A in the homework, I've put together a step-by-step guide that basically walks you through the whole assignment (see HW1_SuggestedWorkflow.pdf). Take advantage of this!

- Ask me or the TA if you have questions. We'll try to reply as soon as possible.

- Make sure that you use the class forum a lot to discuss questions/problems with other students. You are welcome to discuss approaches to problems, etc., just don't give away your source code to others. In the end of the course, I might "bump up" the grade slightly for students that have been participating heavily in the forums, both asking questions as well as providing help for other students.

15. Submission Instructions and Deliverables

You are to write your code for this assignment in C or C++ using **the Visual Studio environment** (probably C++ will be easier to use for this). You may not use another programming language or environment because it will be too difficult for us to grade. However, you are free to develop in another environment, as long as you then migrate everything into Visual Studio and confirm that it works correctly before the final submission.

Make sure that everything we need to compile your code has been submitted, since we cannot waste time trying to compile it ourselves. **This includes the .sln solution file, along with all the .c, .cpp, and .h files so that we can compile your code.** You should modify the SampleScene.cpp to add your own scenes and rendering this should be as simple as running the main() function there.

Zip up the source code and deliverables and submit them on the GauchoSpace before the deadline. You will need to submit the following items by the deadline:

- Source code for your project, including the necessary .sln and .vcproj files that will be needed to setup and compile your project in Visual Studio.
- Scene files and images for your test scenes (part 8) for your "nice scene" you modeled (part 9) . Make sure you include the shaders you wrote. **Please make sure you submit the images, it makes grading your project much easier!**
- Any extra credit you did (part 12).

Late assignments will be subject to a one-letter grade penalty per day late (e.g., an assignment one day late that would normally get a B would get a C), unless you choose to use some of your late days for this assignments.

Best of luck!