Math 155B – Topics in Computer Graphics – Spring 2020

Project #5 – Implement Multiple Distributed Ray Tracing features.

Overview: You will implement distributed ray tracing, with three of the following four features: (a) Anti-aliasing, and (b) Soft shadows and (c) Depth of field and (d) Motion blur. You may choose which three to implement.

Due date: Completed project due Monday, May 25, 10:00pm. Upload a main program and a PDF file to the google drive folder that was shared with you, and a new subfolder named "Project5".

Grading is an individual zoom session with Jon Pham or Professor Buss. Your PDF upload should discuss briefly both technical and artistic aspects of the project. It should include pictures showing your results.

You will continue working with the RayTrace softare from Project 4.

1. This is a large Visual Studio "Solution" consisting of six Visual Studio "Projects". The different projects are:
    a. **RayTraceKd.** This is main project, and the one you will modify for the project. The ".sln" Visual Studio Solution file for the entire solution is in this folder. Open this file to open work with the project in Visual Studio.

       You will modify the two files **RayTraceKd.cpp** and **RayTraceSetup155B.cpp** for your project.

       RayTraceKd.cpp has the high-level code for the ray tracing algorithm.
       RayTraceSetup155B.cpp has the scene description. You may use this mostly as is, but you will also need to modify to update the scene and to better illustrate your distributed ray tracing.
       RayTraceSetup2.cpp has the scene description of the picture on the cover of the textbook. You can see examples of all the features of setting up geometries for the ray tracer.
       RaytraceKdMain.cpp is the main program. You probably do not need to modify this unless you add extra keyboard or mouse controls, or want to change which scene is rendered (with the MODE variable).
    b. **OpenglRender** – intended to give a quick and crude rendering of the scene in OpenGL, but is still only partially ported from legacy OpenGL to modern OpenGL. Minor updates to these features might be rolled out over the rest of the course.You do not need to work with these files for your project.
    c. **RayTraceMgr** – helper functions for setting a geometric scene, or reading .obj or .nff files. You do not need to work with these files for your project.
    d. **Graphics** – handles the main ray tracing calculations of intersecting rays with objects. You do not need to work with these files for your project.
    e. **DataStructs** – includes the KdTree data structure, plus various other useful data structures. (Nowadays partially superseded by STL.) You do not need to work with these files for your project.

   f. **VrMath** – include GlLinearR3, GlLinearR4 and other math packages. You do not need to modify these files for your project.

2. **Building the C++ Solution/Project.** In project 4 you learned how to compile and modify the RayTrace program. It runs faster in "**Release**" mode, but you should definitely do all program development in "**Debug**" mode with small render windows. (Release mode often runs 10-20 times faster.)
3. **Running the program:**
   a. Keyboard controls: **Arrows** control view direction. **Home/End** moves the scene further away or closer. **Arrow keys with Shift pressed** move the scene up and down, and left and right.
   Pressing "**R**" resets the view to its original position.
   b. At first, you will see only crude shapes --- not ray traced.
   Press **SPACE** to start the ray tracing. Then wait a few seconds. Keep the window small most of the time, to keep the raytracing fast.
   c. Whenever you change the view or the window size, it reverts to showing just outline shapes in OpenGL mode. Press SPACE again to ray trace.
   d. Possibly helpful for debugging: a mouse click prints the x,y coordinates to the console window.
4. **Examine source code in RayTraceKd.cpp.**
   a. The routine **RayTraceView** is the top-level ray tracing routine. This is routine you will modify to do anti-aliasing (by subpixel jittering) and depth-of-field (by eye position jittering).
   b. Find the routines **CalcAllDirectIllum** and **ShadowFeelerKd.** These are the routines you will modify for implementing soft shadows.
   c. Find the routines **myExtentsFunc()** and **myExtentsInBox().** These are used as callback routines by when the kd-Tree is created (when ObjectKdTree.BuildTree() is called). These callback routines return an axis-aligned bounding box enclosing a ViewableObject. The latter function intersects the AABB with the AABB of a kd-Tree node.
5. **Examine source code in RayTraceKdMain.cpp.** This is the main program; handles all windowing interfaces, keystrokes, etc. Note where the variable **MODE** is used to control which scene is shown. You will make very minor modifications to this program, perhaps none.
6. **Recall that RayTraceSetup155B.cpp** (and also in **RayTraceSetup2.cpp**) define the geometric objects and lights and materials of two different scenes. One a simplified one (MODE==0), and one as shown on the book cover. You might need to make some additions to **RayTraceSetup155B.cpp** to customize the scene to make your scene look good.
   If you do motion blur, you will need to add an object to the scene that can be moved around.
7. **Your tasks:** There are two aspects:
   Main aspect: Implement three of the following four features: (a) Anti-aliasing, and (b) Soft shadows, and (c) Motion blur and (d) Depth of field. You can receive a little "extra credit" (1 point) if you implement all four.
   Second aspect: Change the contents of the scene, to make it more attractive, and illustrate your distributed ray tracing features. For instance, possibly reposition objects and lights to make the soft shadows more apparent. Possibly reposition objects or resize the scene to make depth of field more apparent.
8. **Anti-aliasing.** In RayTraceView, cast multiple rays to each pixel. Use jittered, stochastic super-sampling. The routine **MainView.CalcPixelDirection** currently takes integers **i, j** as inputs, but these can be replaced by floats to access subpixel locations. (This makes the code modification rather simple, but see 10.b below.)
   It is highly suggested to have a variable (either a #define or a "const" integer) that controls the number of subpixel locations, so you can experiment easily with 2x2 or 3x3 or 4x4 or 5x5 subpixel samples.

9. **Soft shadows.** The suggestion is to add a new parameter to the function **ShadowFeelerKd**, called **displacement** (for instance), which specifies a displacement added to the light's position. You can arrange the displacements in a circular pattern or rectangular pattern. (Rectangular is easier.) If the lights are horizontal rectangles (like ceiling lights), this makes it easier too.
    a. In prior years, I recommended having **CalcAllDirectIllum** call **ShadowFeelerKd** multiple times. (2x2 or 3x3 or 4x4, etc. many times) then find the fraction of rays which are unobstructed.
    b. This year, as in the prerecorded video, I am recommending instead: (i) for each pixel and each light, pick a jittered, stochastic position on the light, (ii) cast only one shadow feeler to the light in each call to CalcAllDirectIllum. For rectangular lights, step (i) can be a fixed position on the light.
    Be sure to use separate random permutations for the lights. (At the very least different from the ones used for anti-aliasing, depth-of-field, or motion blur. It probably makes no visible difference if a single random permutation is used for all the lights.)
    c. You may use either method a. or b. in your implementation.
10. **Depth-of-Field.**
    a. To displace the eye position up and down in **RayTraceView**, use the methods **GetPixeldU()** and **GetPixeldV** (belonging to the **CameraView** member object **MainView**). These give vectors (VectorR3's) giving the distance horizontally and vertically between ***adjacent*** pixels. Use suitably scaled random multiples of these values to jitter the eye position left-and-right and up-and-down.
    b. You can no longer use **CalcPixelDirection** to get the ray direction. Instead, call **CalcPixelPosition**, and subtract the jittered eye position and normalize to get the ray trace direction.
11. **Motion blur.** Jitter the position of some object or objects, either just translation, just rotation, or translation and rotation combined. This may be the trickiest one to implement, since the KdTree by default uses a tight bounding box and this is not valid if the object is moving. (Unless you implement a sphere with only rotation and no translation: and this is acceptable for the programming project!!!)
    a. To transform a general viewable object, use **TransformWithRigid** as defined in **TransformViewable.h**, using a **RigidMapR3** as defined in **LinearR3.h.**
    b. One way to make the object be handled correctly by KdTree if the object is changing position: Modify the routines **myExtentsFunc** and **myExtentsInBox** in **RayTraceKd.cpp** to check for the object number (objNum) of the moving object, and return a large box containing all possible positions of the moving object instead of calling **CalcAABB** or **CalcExtentsInBox** for that object. The object number is the ViewableObject's index as added to **TheScene2** in **RayTraceSetup155b.cpp.** The class **AABB** is defined in **AABB.h** in the **VrMath** project. You set an AABB by specifying its minimum values for x,y,z, and its maximum values for x,y,z as two VectorR3's. Take the intersection of two AABB's using the method **IntersectAgainst** from the AABB class. (Please ask Jon Pham or Professor Buss or – even better – post publicly to piazza, to get help with coding this.)
    Another alternative to all this is to test for an intersection of the ray with the moving object in all cases. In this latter case, the moving object should not be in the kd tree.
    c. To repeat: you can skip all the work in paragraph b. if you use a spinning sphere, since its AABB does not change with the animation.
12. **Adding new objects or features to the scene.** You learned how to do this in Project 4. See the code in RayTraceSetup2.cpp for examples. You should add or move items in the scene as needed to illustrate the advantages of the motion blur, the soft shadows and the depth of field.

Hand in: Upload to the google drive shared folder, in a subfolder "Project5":

(A) By Wednesday evening, May 20: A PDF with one or two images showing your progress so far. These will be discussed in class on Thursday. You may hand these in via the google drive folder, or post them publicly to the piazza web pages (anonymously is OK), preferably both.

By Monday, May 25 due date:
(B.i) A PDF file showing two or more images of your scene, and give a description of the scene and its features. The description of the features should include descriptions of both technical aspects and artistic aspects of the scene. Include screen images of "before" and "after" pictures showing your scene with and without distributed ray tracing techniques. Your pictures should make it apparent how the distributed ray tracing has helped. (If they do not help, they probably have not been properly implemented.)
Length: 2-3 pages, including pictures.
(B.ii) Your main program RayTraceKd.cpp. And any other code you think will help us evaluate your work.

Grading will be based on technical merit, creativity, and artistic merit. It most cases, technical merit will be the primary criteria, and is sufficient for full credit. But, artistic merit can help compensate for missing technical aspects.