EECS 351-2: Intermediate Computer Graphics **Project B: Ray-Tracing Platform**
Instructor: Jack Tumblin Spring 2018

Project Demo Day (and due date): Thurs June 7, 2018

Write a hybrid WegbGL/ raytracing program that shows, side-by-side, two realistically-lit pictures of the same 3D scene: an interactive WebGL preview on the left, and a high-quality ray-traced version of the previewed scene on the right. To trigger rendering of a new ray-traced image, users will press the 't' or 'T' key to start the ray-tracing process.

Requirements:

A)--In-Class Demo Day: when you will demonstrate your completed program to the entire class, and to two other students that will each evaluate your work by completing a 'Grading Sheet'. You then have about 3 days to revise and improve your project with what you learned from Demo Day before you submit your final version for grading. The grading sheet markings plus all the improvements we find in your finalized project determines your project grade.

B)-- Submit final version on CANVAS no more than 72 hours +1 weekend-day later (Monday June 11, 11:59PM) to avoid late penalties. Submit just one single compressed folder (ZIP file) that contains:
i) your written project report as a PDF file, and
ii) (mimic the 'starter code' ZIP-file organization). We must be able to read your report & run your program in the Chrome browser by simply uncompressing your ZIP file, and double-clicking an HTML file found inside, in the same directory as your project report.
--- Name your ZIP file and the directory inside as: `FamilynamePersonalname_ProjB`
For example, my project B file would be: TumblinJack_ProjB.zip. It would contain sub-directories such as 'lib' and files such as TumblinJack_ProjB.pdf (a report), TumblinJack_ProjB.html, TumblinJack_ProjB.js ,etc.
---To submit your work, use CMS/Canvas→Assignments. DO NOT e-mail your project (not accepted!).
---BEWARE! SEVERE LATE PENALTIES! (see Canvas→ … SyllabusSchedule….pdf)
---BEWARE! Limited late-time! Registrar requires me to submit your final grades by end-of-day Sunday, Jun 17…

Project B consists of:

1)—Report: A short written, illustrated report, submitted as a printable PDF file.
Length must be at least 1 page and is typically <6 pages, but you should decide how much is sufficient.
A complete report consists of these 3 sections:

a)-- your name, netID (3 letters, 3 numbers: my netID is jet861), and a descriptive title for your project
(*e.g.* "Project B: Checkerboard Hall of Spherical Mirrors," not just "Project B"), and

b)-- a brief 'User's Guide' to explain your project, with complete instructions on how to run and control the program (e.g. A,a,F,f keys move camera left/right; arrow keys rotate camera left/right and move forward/backward; pgUp/Dn tilt camera up/down, S,s,D,d keys adjust Perlin Noise parameters,…).
CAUTION: Your classmates should be able run, understand and extend your project from this report alone.

c)-- a brief, illustrated 'Results' section that shows **at least 5 still pictures** of your program in action (use screen captures; no need for video capture), with figure captions and text explanations.

2)--Your Complete WebGL Program, which must include:

a)--On-screen User Instructions: Your program must always easily explain itself; it should never puzzle its users. Users must be able to find help easily, without reading your report or asking you for help. How? You decide! Perhaps you could put instructions on the webpage outside the 'canvas' element, or in a pop-up CSS window, or add a 'HELP' button, or use the WebGL book's 'HUD' method (pg. 368), or on-screen text: 'press F1 (Help) key to print instructions on console', *etc.* Print brief but complete and specific instructions; *e.g.* "press arrow keys to move camera, drag the mouse on WebGL preview to rotate the camera, press +/- keys to move the robot's arms, < or > to move legs."

b)—Side-by-Side WegbGL/Ray-Traced Image Display. Your program, like the starter code, presents users with two images side-by-side. On the left, users see an interactive WebGL-rendered preview they can modify with

keyboard and mouse to create a pleasing arrangement of camera, lights, and shapes. After arranging the scene as they like it in the WebGL preview, users press the 't' or 'T' (trace) key to compute and show a matching ray-traced rendering of that scene and display it on the right. Your ray-traced image must depict the 3D scene of your WebGL preview.

c)--Matched WebGL and Ray-Traced Scenes. Your program must show an exact geometric match between WebGL and ray-traced images for any camera position, any viewing direction, and any scene or arrangement of shapes, materials, and lights. While your ray-traced image will surely offer better looking lighting, materials, and effects (*e.g.* antialiasing, shadows, reflections, transparency, refraction, 3D solid textures, CSG), it must depict the same basic shapes, light-sources, and cameras in exactly the same positions and orientations. For example, your WebGL preview will draw a sphere using triangles that only approximate the perfect sphere drawn by your ray-tracer, but that sphere's on-screen position, radius, and scaling must match exactly.

d)—Vast 'Ground Plane' Grid: Your program must clearly depict a vast, seemingly endless horizontal 'ground plane': a square grid of thick lines on a different-colored background fixed to the **x,y plane** of your 'world-space' coordinate system **(DO NOT create your ground plane in the x,z plane!)**. This grid should make any and all camera movements obvious on-screen, and form a reliable 'horizon line' when viewed with a perspective camera.
HINT: In the camera or 'eye' coordinate system, the u,v plane is vertical, aligned with 2D image coords x,y. In the 'world' coord system, the x,y plane is horizontal: z axis points up to the sky. I strongly recommend positioning WebGL camera with a 'setLookAt()' function, and an equivalent rayLookAt() function member for your ray-tracing camera.

e)-- **3D Interactive Viewing**: users must be able to move & aim the 3D camera in 3D. You must demonstrate how to aim your camera in any desired 3D direction (including tilting up/down) without changing its 3D position, and then move in that aiming direction without changing the aiming direction, and also 'strafe' horizontally, perpendicular to that aiming direction(e.g. apply the 'glass tube' analogy).

f)--User-Adjustable Antialiasing: Allow users to select between no antialiasing (one tile-centered ray per pixel) and antialiasing done by jittered super-sampling (demo day: must show ***at least*** these two user-selectable cases:
1 centered sample per pixel with no jitter, and 4x4 jittered super-samples/pixel. Better: selectable 2x2,3x3,4x4,5x5, etc.)

g)--At Least 2 Scenes. As you are designing a ray-tracer that can depict *any* 3D scene, your program must let users select between at least 2 wholly-different scenes on Demo Day, without modifying code or re-starting the program. These scenes must contain different shapes and/or assemblies of shapes (e.g. jointed objects?) of different sizes at different locations and orientations and made of different materials. Specifically, each scene must have:

h)--At least 3 Transformed Spheres. As discussed in class, your ray tracer must transform rays from 'world' coordinates to 'model' coordinates before tracing. Instead of writing code to find ray/sphere intersections for spheres at a specified x,y,z location and specified origin, write code to find intersections of a ray **transformed** by a matrix (e.g. worldRay2model) with a unit-radius sphere at the model-space origin. The matrix contents' effect on the rays cause the ray-traced sphere to appear in world space with any desired translation, rotation, and scale (including non-uniform scale that forms a long ellipsoid or flat 'pancake' shape, just as the EECS351-1 starter code scaled cubes to make robot arms).

j)--At least 3 other Transformed Shapes of at least one other, 3rd kind. Spheres on a ground-plane are not enough! Create at least one other kind of shape, such as a cube, a cone, a cylinder (try a different radius at each end), a torus, a tetrahedron, a dodecahedron, or even super-quadrics, meta-balls or blobbies; use at least 4 non-spheres in all scenes.

k)--Lighting & Shadows. Your ray-tracer must include at least two user-positioned, user-switchable (on/off) light sources that can cast shadows correctly from any shape to any other shape in the scene. Show that all shadows combine properly when overlapped: each shadow darkens the other in a visually plausible way, and switching each light on/off changes those shadows in the visually-correct manner. Be sure you can move light-sources easily and independently—this will help you verify that your ray-tracing light sources are correct by their match to the WebGL preview's light sources. If you wish, one of your movable light sources may be a 'headlight' attached to the (user movable) camera.

m)--Phong Materials. Your ray-tracer must recreate the basic Phong lighting method (not Blinn-Phong!) used in your WebGL preview window. As you'll recall, these materials use ambient, diffuse, specular reflectance color values and

an optional emissive color (R,G,B for each), and require light sources with ambient, diffuse, and specular terms (also R,G,B for each) to compute the color returned from a surface point. Use floating-point values for all of these quantities, and store final values for pixels as floating-point RGB (see 'CImgBuf' object in Week01 starter code).

n)--Mirror-Reflection and Recursion. Unlike WebGL, your ray-tracer must be capable of recursive reflections created by spawning additional rays at ray/surface intersection points (e.g. in findShade() functions), as demonstrated in class. Users must be able to adjust the maximum depth of this recursion, which determines how many inter-reflections the resulting image may depict -- 0, 1, 2, 3,4, … N. For most scenes, recursion depths beyond 3 or 4 offer only subtle, hard-to-see improvements in the image, but you program must allow users to explore any desired recursion depth from 0 to 16. For example, if you construct a 'hall of mirrors', users should be able to see the result of at least 16 repetitive reflections if they set recursion limits high enough and specular reflectances high enough (HINT: try 1.0).

Outside Sources & Plagiarism Rules:

Simple: *never* submit the work of others as your own. You are welcome to begin with the book's example code and any current or old 'starter code' I supply; you can keep or modify any of it as you wish without citing its source. For any graphics program I strongly encourage you to start with a basic graphics program (hence 'starter code') that already works correctly, and incrementally improve it, testing and looking at image results at each step. Don't make massive untested changes all at once; you may never find all the flaws hidden in all the new code, and it's never easy to test it thoroughly. Instead, break down your big changes into small easy-to-test steps; if one small step ruins the program, search only your small set of most-recent changes and you will always find the fatal flaws. Also, please learn from websites, tutorials and friends anywhere (e.g. .gitHub, openGL.org, google 'webGL raytrace' 'javascript raytrace' etc), but you must always properly credit the works of others—**no plagiarism!**

Plagiarism rules for writing essays apply equally well to writing software. You would never cut-and-paste paragraphs or whole sentences written by others unless they were clearly marked quotes with fully-cited sources. The same is true for whole functions, blocks and statements written by others. Never cut-and-paste others' code without crediting them, and never waste time trying to hide it by rearrangement and renaming (MOSS won't be fooled). Instead, write your own code to gain new sound principles well-learned and get good grades well-earned. Study closely others' good code, learn its best ideas, and then close the book and the website. Take the good ideas, but not the code: add a gracious link/citation/URL to the inspiring source of the ideas, and then write your own, better code in your own better style. (If I find plagiarism evidence, the University requires me to report it to the Dean of Students for investigation).

Weekly Goals:

Remember, these are ***minimum*** weekly goals, assignments imposed to help you avoid accumulating a manic tragedy the end of the quarter. To do well in the course, and for the most pleasant and productive progress on your ray-tracer, try to stay ahead of these minima. Give yourself time to explore, to invent, and to investigate any quirks you find. Don't worry if your images seem too simple; nearly all of the most visually interesting and most delightful, noteworthy results tend to appear later, piled up as you approach the Week 4 Goals.
Reaching the Week 4 goals before week 4 ensures you have more time for the 'good stuff'!

Week 1: First Ray-Traced Image
Create the 'skeleton' of your Ray-Tracer with WebGL preview.

- Side-by-Side display: WebGL preview on left, ray-traced result on right (see Week1 starter code).
- Camera (defined at origin) looks at xy grid-plane in cam coordinates;
  grid has adjustable z-value depth.
- Set up most-basic ray/plane intersection (see Week1 'supplement' starter code + lecture notes)
- Do not attempt any ray transformations yet (don't try to tilt the grid-plane in world coords!)
- Organize: create 'stub' functions and prototypes for camera, ray, hit, hitlist, shape, material, etc. (see Week1 'supplement' starter code for an example. Use these, or make your own)

- Construct rayPerspective() and rayFrustum() functions to specify ray-tracing cameras; use the same arguments needed for WebGL's gl.perspective() and gl.frustum(), and construct a geometrically-equivalent ray-tracing camera.
- Construct a formal camera-positioning function as part of the ray-camera object (e.g. CCamera.rayLookAt(), using the same arguments needed our WebGL setLookAt() function.
- Test it -- can you ray-trace your 'grid' object? Can you move the ground-plane's z value and see a correctly-ray-traced result?
- Experiment! -- Pivot the camera by 90 degrees in world-space to see linegrid horizon.

Week 2: Anti-Aliasing + Transform Matrices for Ray Shapes

- Add user-adjustable antialiasing from within camera prototype; implement jittered super-sampling in the ray-tracing camera (see Lecture Notes C on aliasing/antialiasing).
- Add ray-transformation-making member functions to your shape-describing class (e.g. 'CGeom'): rayLoadIdentity(), rayTranslate(), rayRotate(), rayScale(), etc. to let your ray-tracer's image exactly mimic the WebGL preview image by function calls with identical arguments to render in WebGL and by ray-tracing.
- Add worldRay2model transform matrix as member of shape-describing classes, and apply it to rays before finding ray/object intersections. Try it! Can you translate, rotate, and scale your grid-plane without moving the camera?
- Add spheres to your shape-describing class(es), but draw them using uniform color. Do you get a uniform disk color? Next, set color using surface normal's z-coordinate: z<0==black, z>0==white…
- Test worldRay2model transform matrix: can you 'squash' a sphere to make a pancake-like shape?

Week 3: Simple Lights & Basic Materials (Lights, Reflectance & Shadows with code well-organized for recursion)

- **Light Source objects** (CLight); Add a light-source-describing class; start with point-light sources using Phong light parameters (ambient, diffuse, specular illumination, etc).
  Start simply (e.g. diffuse-only), test it, then add more parameters later.
- **Materials objects** (CMatl): Add Phong materials-describing class (ambient, diffuse, specular, emissive). Start simply (e.g. diffuse-only), test it, then add more parameters later.
  --Be careful to keep *all* materials descriptors separate from shape-describing objects, and instead let shapes each keep a name or an array index to select among a list of available materials objects.
  --Materials selectable by index# or by name provides an easy way for each shape object to select from a list many materials. It also allows multiple CGeom objects to share one material-describing object, and enables you to devise shapes made from two or more materials (e.g. 2-material checkerboard shape, etc.).
- **Test ray-traced shading**: put one light source at the eye-point, aim camera at center of a sphere: a perfectly round specular highlight should appear at the exact center of the sphere.
  If not, something is wrong...
- **Test camera positioning** and its on-screen effect on lighting & materials.
  How? Place the light source at a position well away from the camera; move the camera or light source and that specular highlight should move across the surface of the sphere, but when you rotate the sphere the highlight should NOT move. If it does, you probably did some of your lighting calculations in 'model' or 'camera' coords instead of 'world' coords -- did you transform your CGeom shape normals to 'world' coords? Did you compute 'view' and 'reflected' rays in world coords?
- **Test surface-normals transformations**; scale a sphere along just one axis to 'squash' or 'stretch' it. As you move the light (but not the camera), does it correctly move the sphere's specular highlight?
- **Shadows:** add shadow-ray tests. **CAREFUL!** shadows 'turn off' only the diffuse and specular terms in the Phong lighting model; ambient term still lights the shadowed surfaces.

Week 4: Recursion + Complex Scenes
Multiple inter-reflections, transparency; computing procedural materials, soft shadows and more

- Add more objects to your scene; be sure you can apply ray transformation functions to ray-traced shapes; squash/stretch them, rotate them, make jointed objects?

- Build more complex shapes using transformation trees, built with identical calls to WebGL drawing-axis transform functions and their equivalent ray-transform functions (e.g. call both glRotate() and rayRotate() functions using identical arguments, etc.).
- Add mirror-reflectance colors to surface materials. Add reflected-ray tracing to your findShade() function: extend to recursive mirror reflections with user-controlled recursion depth.
- Experiment with procedural textures: can you make a 3D checkerboard of 2 selectable materials? can you make procedural 'bump maps' too (position-dependent perturbations to the surface normal)?
- OPTIONAL: Stochastic sampling for soft shadows (**Peter Shirley 'Ray-Tracing in a Weekend' $3 e-book**)
- OPTIONAL Perlin noise? Turbulence functions? Texture-maps? Synthetic wood (cylindrical sinusoids)?
- OPTIONAL: transparency; add materials members for index of refraction; create new 'transparency' rays recursively, guided by Snell's Law (lecture notes)
- OPTIONAL: Area-light sources to create soft shadows. Replace point-source with subdivided or randomly-sampled area or volume. For a rectangular area light, sub-divide the rectangle into small equal-area pieces and shoot a 'shadow' ray to each one of them. Set shadow strength to the ratio of blocked/unblocked shadow rays. (question: would area light have cosine falloff? See Week 1 Lecture Notes on measuring light).
- OPTIONAL: Newton-Raphson solver to find ray/object intersections by root-finding (when no analytical solution exists). See Lengyel Week 1 reading. Try it first on Blinn blobbies, shapes, try Bloomental shapes, etc.
- OPTIONAL: better-than-Phong lighting/shading: Cook-Torrance, He, LaFortune or others.
- OPTIONAL more shape primitives: generalized cylinders? torus? super-quadrics? (google it)
- OPTIONAL: Constructive Solid Geometry: implement the 3 basic set-operations: Union(+), Difference(-) and Intersection(*) on 3D shapes to create novel shapes from simple primitives. For example: intersection of 2 overlapping spheres form a lens; Cylinder union with 2 spheres of same radius makes dome-like end-caps; difference of box and cylinder drills a hole in the box.

How far can I advance this Ray-Tracer?

Your ray-tracer's design holds all the essentials of a modern full-featured general-purpose renderer, and thus it offers a huge range of possibilities for further improvement well beyond what we can do in this class. **Explore it; Google/Bing!**

You can extend your ray-tracer to unlimited exotic shapes by writing robust 1-D 'root-finding' code to find ray intersections with arbitrarily-defined implicit functions. Ray-tracing by root finding takes more time but can vastly extend scene contents and visualize exotic evolving shapes, such as Blinn blobbies and 'meta-balls'; superquadrics; fluids and smoke; skeletons skinned with distance functions (Google/Bing: 'Jules Bloomenthal' 'unchained geometry').

Higher-dimensional procedural and noise-like functions can add interest and extend visual complexity to both shapes and materials, beginning with '3D textures' or 'solid textures' and extending to Perlin Noise.

Implementing Constructive Solid Geometry (CSG) discussed in the last parts of FSHill Chapter 14 reading, enables your ray-tracer to perform set operations between any and all implemented shapes to yield astonishing results. With a bit of exploration of stochastic methods, your finished ray tracer will then have most of the core components you need to pursue physically-accurate simulation of optics, materials, lighting, and light transport (see 'path tracing', 'radiosity', 'global illumination', 'photon maps' etc). If the course were longer, I would first ask everyone to add a general-purpose root-finder to their ray-tracers, then have us test it using James Blinn's 'Blobbies' or 'meta-balls'. Next urge you to implement CSG, and then extend findShade() to glass and transparent materials to enable simulation of lenses, followed by implementation of Perlin noise & turbulence for materials and shapes. But you might have other priorities and preferences: follow them first in Project B, and work to explore tracing features you find most interesting.

Don't worry if you ray-tracer seems slow; you can be certain that it will get even slower as you add more interesting features. Many people expended (and expend) massive efforts on performance improvements for ray-tracing since its inception (Whitted; 1979), first done algorithmically (BSP trees, bounding-bo hierarchies and ordered intersection tests, spatial, geometric & temporal cache methods; adaptive ray positions that trace outlines, visibility preprocessing, cone-beam tracing and more), then through parallelism and specialized hardware (ray-tracing is embarrassingly parallel, and often explored on multiprocessor CPUs and GPUs), and recently even nVidia's recent Optix API that ray-traces on the GPU. You can speed up your ray-tracer by any of these methods, but I recommend you don't do that until later – after you implement all the features you might ever want or need. Optimizing a ray-tracer for speed too soon frequently will horribly complicate adding new optical features. Instead, spend your time adding features, not rendering speed!