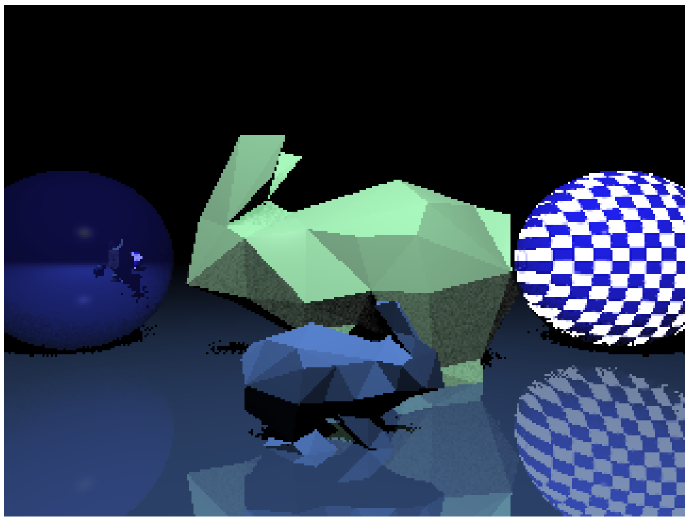
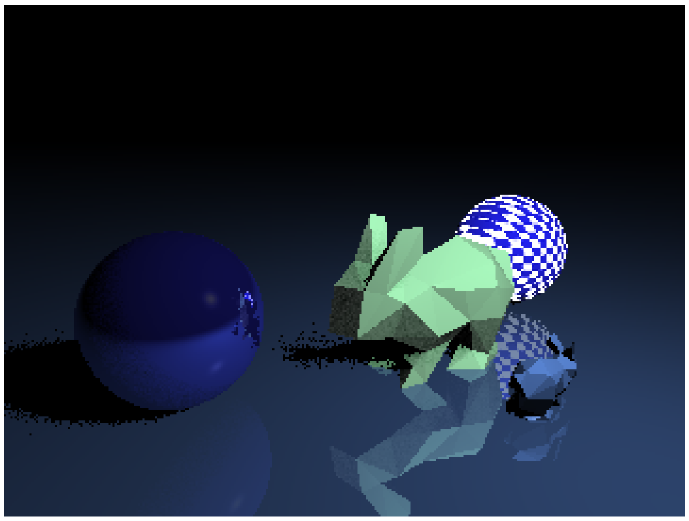
CSC 2504 - Project: Advanced Ray Tracer

Ryan Primeau & Stacey Oue

Anti-Aliasing

- **AA_ENABLED** – flag in *raytracer.h* to turn anti-aliasing on/off
- **AA_SAMPLES** – number of samples used in anit-aliasing

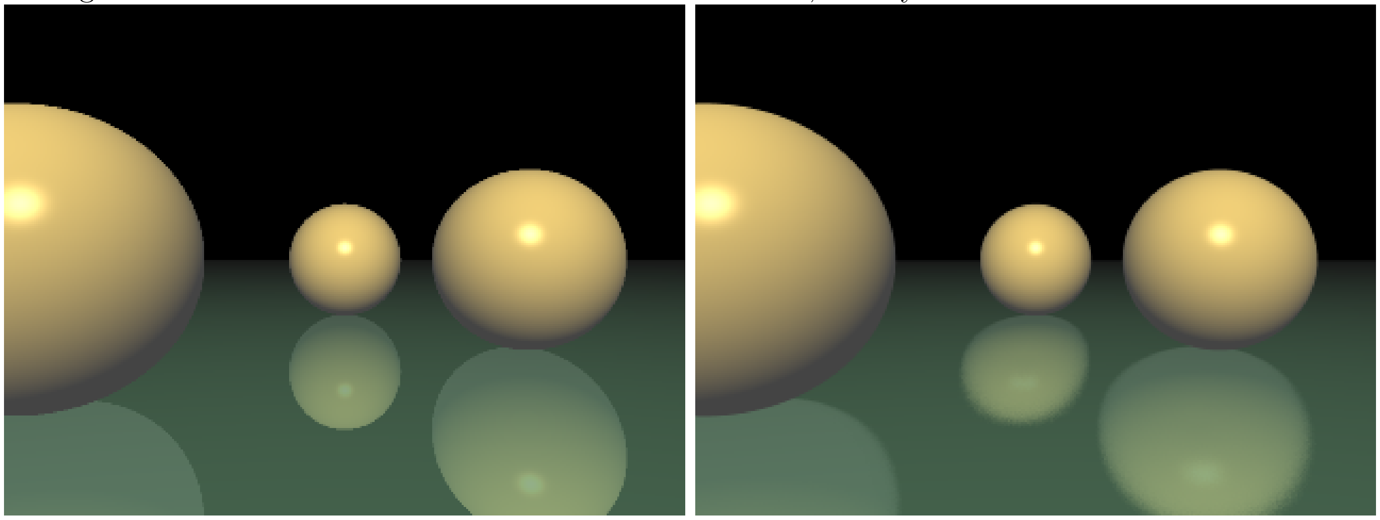
Supersampling, a spatial anti-aliasing method, is implemented to remove jagged and pixelated edges (i.e. *jaggies*).

Glossy Reflection

- **MAX_RAY_DEPTH** - maximum number of light bounces allowed.
- **REFLECTION_ENABLED** - enable/disable reflection.
- **GLOSSY_RAYS** - number of rays casted for reflection (1 for perfect mirror reflection).

Glossy reflection is an extension of reflection implemented in **Raytracer::shadeRay**. A single ray casted creates a perfect mirror reflection; glossy reflection casts multiple rays per reflection to generate a blurred look. The direction of each reflected ray casted is jittered to create the blurred effect.

Figure 1: Perfect mirror reflection - GLOSSY_RAYS=1, Glossy reflection GLOSSY_RAYS=15

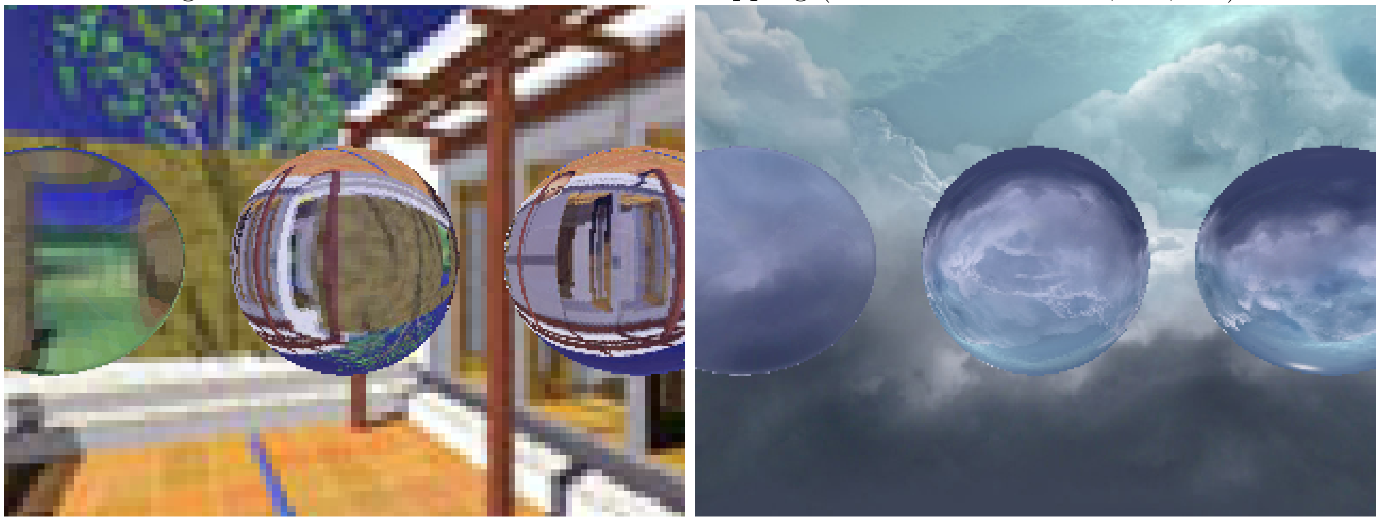


Refraction

- **REFRACTION_ENABLED** - enable/disable refraction.

Refraction is implemented in **Raytracer::shadeRay** and is computed if the object's opacity is not opaque (Materials have an **opacity** property, 0 for opaque and 1 for completely translucent). Index of refraction is computed by determining if the ray is coming from air to object, or object to air and uses the materials **light_speed** property. Refractive constants are defined in *raytracer.h* (i.e. glass is 1.5, air is 1.0 and etc.). The direction of the refraction is computed based on the incoming ray and the material's refractive index, which is used to cast the refracted ray.

Figure 2: Refraction with environment mapping (refraction index: 1.1, 1.5, 1.3)



Soft Shadows

- **SHADOWS_ENABLED** [light_source.h] - enable/disable shadows.
- **SHADOW_RAYS** [light_source.h] - number of rays casted from light source to a pixel (1 for harsh point light source shadowing).

Shadows are implemented in **Raytracer::computeShading** and **LightSource::shade**. Determining if a pixel is in shadow is done by casting a ray from the object to the light source, while checking if there is another object that intersects this ray.

The point light source creates harsh shadows in which either a pixel is in complete shadow or not in shadow at all. Two subclasses of LightSource was added, **AreaLight** and **SphereLight**, to render soft shadows. The area light is implemented as a planar shape, defined by four points and a normal. Sample points are taken from the area light source and are casted to the point. The number of rays that reach the point (no obstructions) are summed up and divided by the total number of rays casted. This percentage is used to determine how dark the shadow is. The sphere light works the same as the area light, sampling is done according to the spherical shape.

For variety the square area light source provides full illumination whenever an intersection occurs and the sphere light uses a an averaging function. When shading the sphere light source divides the number of times the random points generated had line of sight of intersection point on a ray with the max number of attempts. The averaging function has the advantage of smoothing the brightness of the colour and shape of the shadow.

Figure 3: Point light - SHADOW_RAYS=1

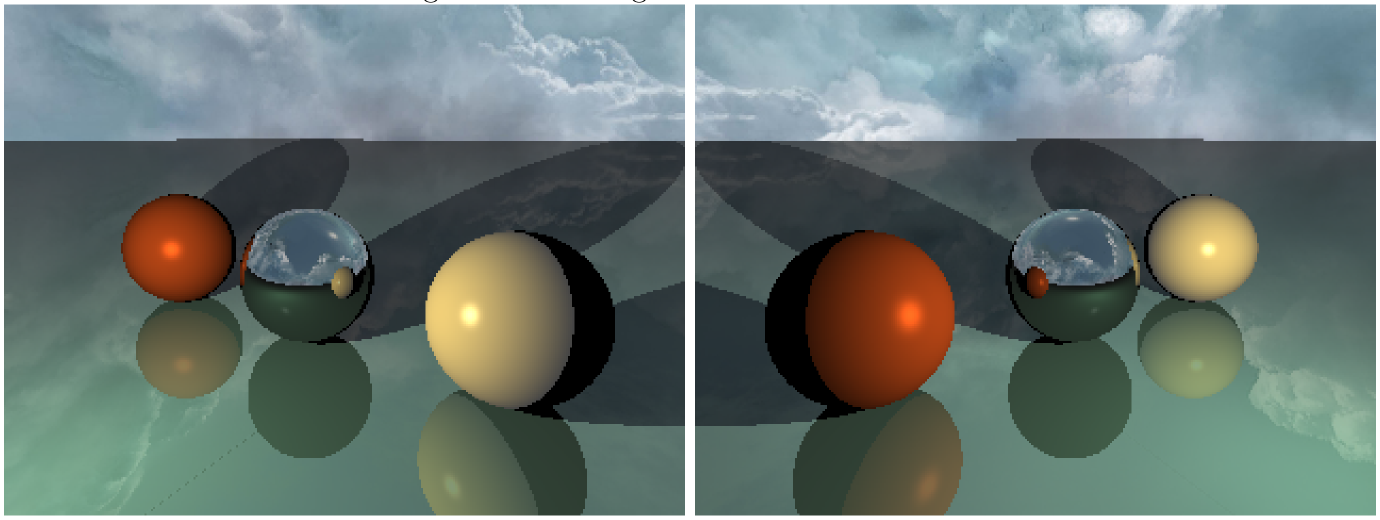
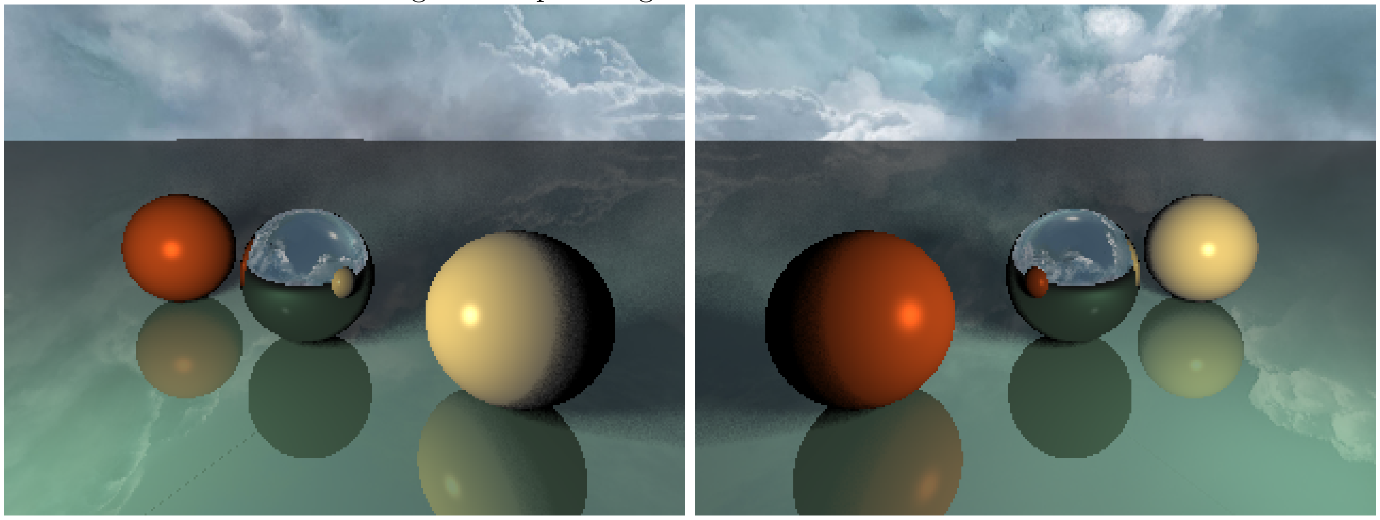


Figure 4: Sphere light - SHADOW_RAYS=10

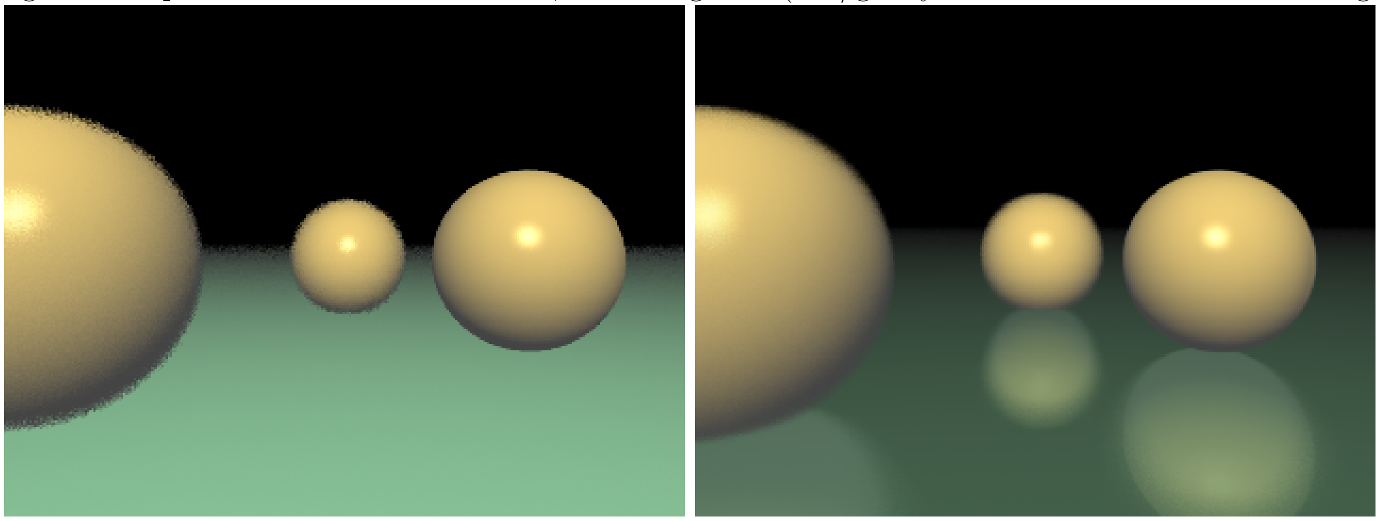


Depth of Field

- **DOF_RAYS** – number of rays casted per pixel with jittered origin (setting to 1 is equivalent to disabling depth of field).
- **FOCAL_LEN** – distance from the camera for which objects are focussed.

Depth of field is computed in **Raytracer::render**; multiple rays are casted for each pixel in which the origin of the casted rays are jittered to create a blurred effect for all the objects that are not within the focal length.

Figure 5: Depth of field - DOF_RAYS=15, Focal length=3 (AA/glossy reflection turned in second image)

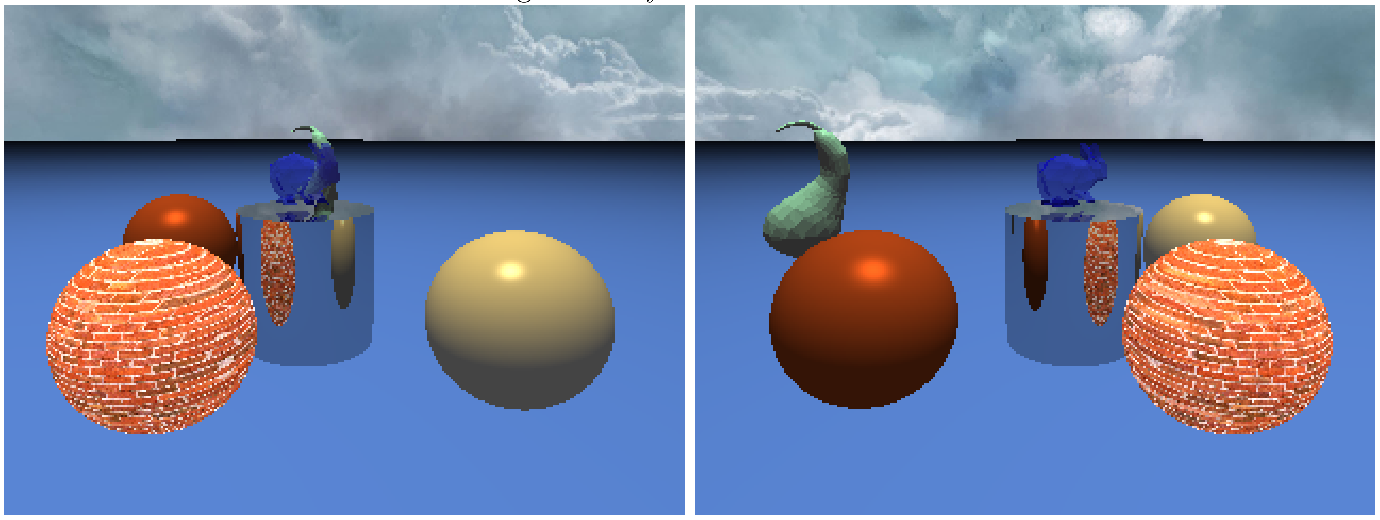


Cylinder

A subclass of SceneObject, **UnitCylinder**, was added to **scene_object.h**. The function **UnitCylinder::intersect** was implemented to determine if a ray casted intersects the cylinder (and affinely deformed cylinders).

Implementation is similar to the unit square and sphere. First, intersection with the infinite cylinder is checked by solving the quadratic equation (as done for the sphere). If no intersection found, the two cylinder planar caps are then checked for intersection (as done for the square but using the equation of the circle).

Figure 6: Cylinder as a mirror

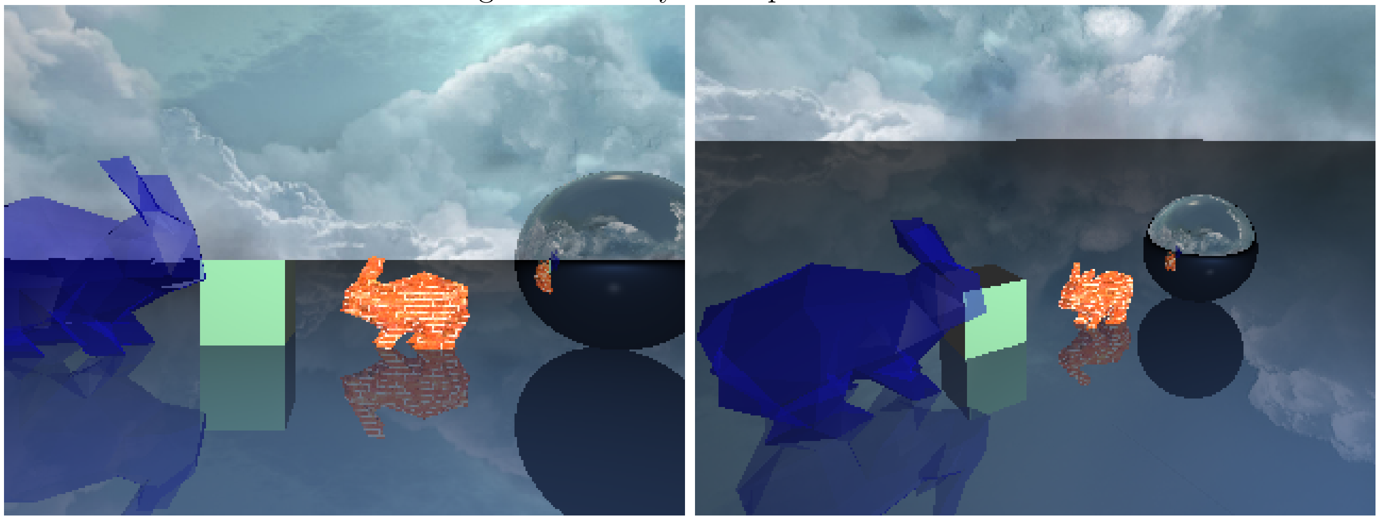


Arbitrary Surface Mesh Geometry

Handling arbitrary mesh objects is done by creating a new class **Mesh**, which reads in *.obj* files and saves the vertices, normals and faces of the object. Each mesh object has a bounding volume to accelerate ray tracing time. A box is put around the mesh object and intersections are first checked with the bounding box, if no hit to the box, then no intersections are checked for the mesh object's triangles.

Additionally, subclasses of SceneObject are created, **MeshObject** and **Triangle**. The MeshObject is used to create Triangle objects associated with the mesh, which are used to determine intersection with the mesh. If the mesh object contains quadrilateral faces, they are handles by being split into two Triangle objects. For example when a MeshObject is added to the scene, Triangles are created from the faces of the MeshObject and added as children to the MeshObject. If a face is a quadrilateral, two Triangles get created and added.

Figure 7: Bunny and square meshes

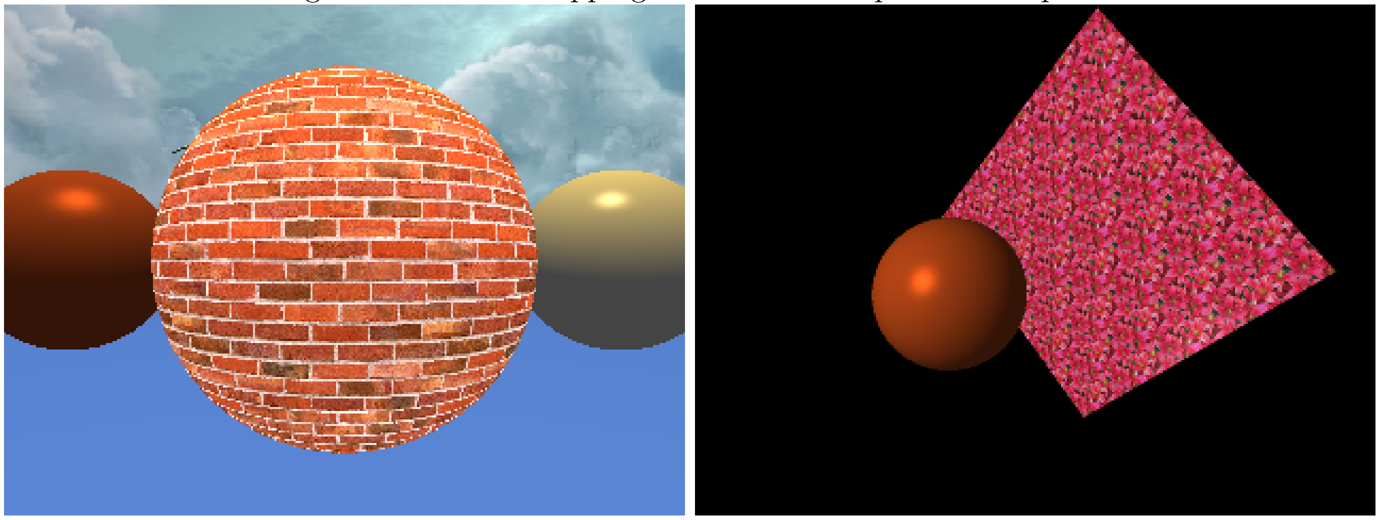


Texture Mapping

A new **Texture** class is added and defined in *util.h*. The functions used to read in the texture files and get applying textures are found in *util.cpp*.

Every material can have a texture. The textures come from an external .bmp file. The bmp file has value for each pixel colour red, green, and blue. We load a single dimensional array for each colour. When an intersection with an object that has a texture occurs we map the point of intersection into the pixel buffers. When shading we append the phong shading results with the colour returned for the texture colour mapper.

Figure 8: Texture mapping - brick and leaf patterns on plane



Environment Mapping

We reuse the mapping colour function from regular texture mappers component. We needed to extended the functionality of the previous function. This part works by applying textures to rays that have no intersection with any object in the scene. We use the most dominant component in normalized ray direction to consider where to map ray into the set left face, back face, right face, front face, up face, down face. We then use the remaining components to map into the standard texture mapping with offsets into the appropriate place in the external cube map .bmp file.

Figure 9: Environment mapping with reflection - villa

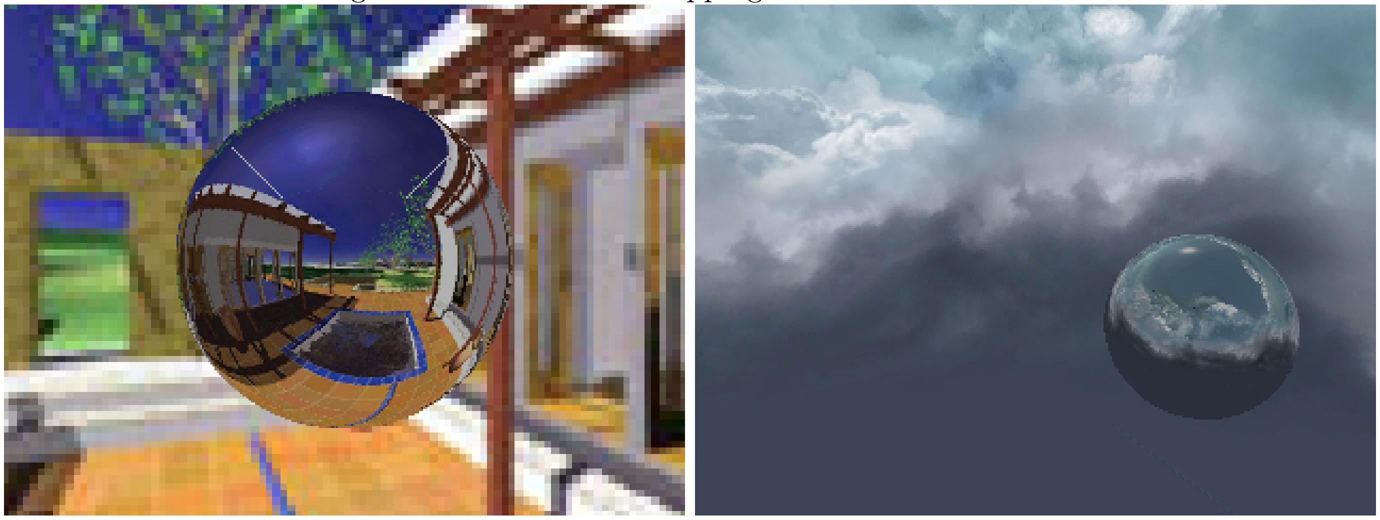
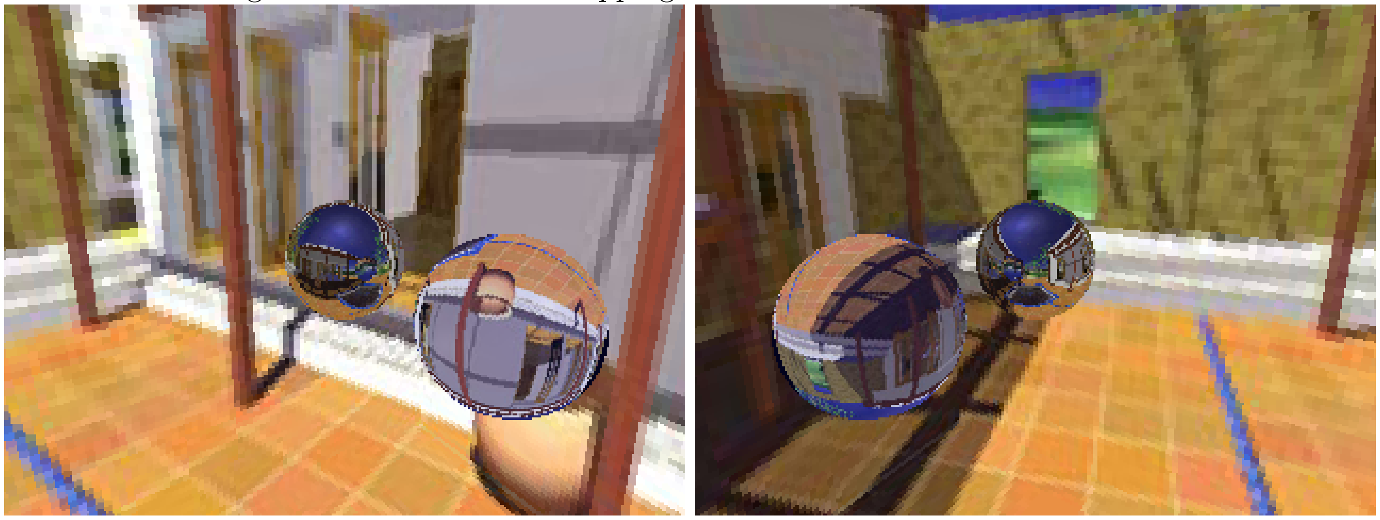


Figure 10: Environment mapping with reflection and refraction - villa



Motion Blur

- **MOTION_BLUR_CYCLES** – number of time frames in the motion blurred image (set to 1 for no motion blur).

Motion blur is implemented in the **Raytracer::render** function. An image is rendered multiple times and any object that has the motion blur property set is translated further in each cycle. To create a blurred motion effect, the color of the pixels of the moving objects are weighted according to the motion blur cycle number.

We experimented with different weighting function but settled on using a finite geometric series with a = 2. The number of terms was generally 5. We would rerun the entire scene multiple times and apply motion each time. The pixel buffer is initially set to 0 for everything and we append the results for each turn multiplied by the weighted value for that turn. The first scene is multiplied by 1/32, the second is multiplied by 2/32, the third is multiplied by 4/32, the fourth is multiplied by 8/32, the fifth is multiplied by 16/32. Once we began to use this weighting function we could begin to actually perceive the direction of the motion.

Figure 11: Motion blur - MOTION_BLUR_CYCLES - 5