CS7GV3 Realtime Rendering: Assignment 5
An Approximate Image-Space Approach for Interactive Refraction

Name: Michael Millard
Student ID: 24364218

April 15, 2025

Abstract

In this project, I investigate real-time refraction rendering by implementing and evaluating the two-surface image-space refraction method proposed by Chris Wyman in his SIGGRAPH 2005 paper *"An Approximate Image-Space Approach for Interactive Refraction"* [1]. The approach aims to efficiently approximate refraction through convex transparent objects by rendering the back faces of the model before the front faces and using a screen-space lookup to determine the exit direction of refracted light. In addition to Wyman's two-surface refraction method I implemented a traditional one-surface refraction approach, as well as a four-bounce path tracer to serve as a ray-traced ground-truth representation of the model in the scene. To assess the performance of each approach, I measured the frames-per-second (FPS) of each method and conducted a visual quality assessment relative to the ray-traced ground-truth image. The one-surface refraction method achieved the best FPS performance with almost all models rendered at over 500 FPS, but visually it lacked physical correctness. The two-surface refraction implementation was consistently between 30-40% slower than the one-surface method with almost all models rendered at over 300 FPS. However, this approach was able to resemble the physically accurate ray-traced version extremely closely and still achieved excellent real-time rendering performance.

Background

This project was based on Chris Wyman's paper entitled *"An Approximate Image-Space Approach for Interactive Refraction"* in [1]. Wyman sought to implement an image-space approach to refraction through two surfaces of convex objects as a feasible real-time alternative to ray tracing, which is highly computationally expensive and only practical with expensive modern-day GPUs.

One-Surface Refraction

Computing refraction through one interface is a trivial practice in computer graphics. Consider the incident ray along vector $\vec{V}$ in Figure 1 below.

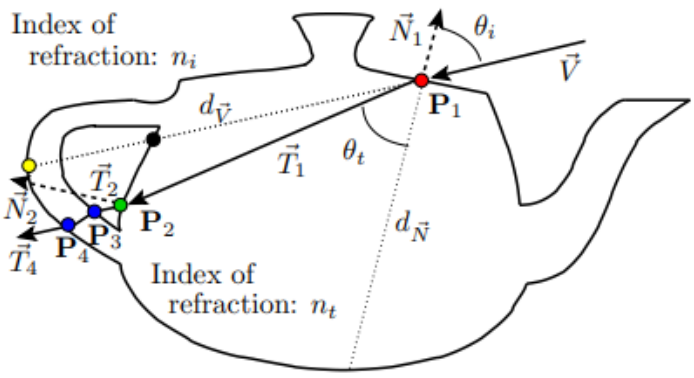

Figure 1: Illustration of refraction through a teapot [1]

The incident ray intersects the 3D model at point $\mathbf{P}_1$, which has a surface normal $\vec{N_1}$. The angle between the incident ray $\vec{V}$ and the surface normal $\vec{N_1}$ is simply computed by taking the dot product of these vectors and isolating θ_i as follows,

$$\theta_i = \arccos(\vec{V} \cdot \vec{N_1}). \tag{1}$$

Assuming the indices of refraction (IOR) of the incident media (η_i) and the transmission media (η_t) are known, the refracted angle θ_t (from which we obtain the refracted vector $\vec{T_1}$) is easily computed using Snell's Law, which states that

$$\eta_i sin(\theta_i) = \eta_t sin(\theta_t). \tag{2}$$

In a one-surface refraction approach, once we have the refracted vector $\vec{T_1}$ along which the light bends inside the transmission medium, we simply sample the environment (in this project, a cubemap skybox) along $\vec{T_1}$ to determine the colour of the corresponding fragment.

Reflection Using Fresnel-Schlick

The approach above assumes a purely refractive model. In this project, since I strictly worked with translucent 3D models that both refract and reflect light, I also implemented reflection according to the Fresnel equations. A popular method of modeling the Fresnel effect in computer graphics applications is to use the Fresnel-Schlick approximation described by Majercik in [2], which approximates the Fresnel term as

$$F_i = F_0 + (1 - F_0)(1 - \cos(\theta_i))^5, \tag{3}$$

where F_i is the Fresnel term that quantifies the magnitude of reflection, F_0 is the base reflectance of the model material, and θ_i is the angle of the incident ray relative to the surface normal. The base reflectance F_0 is simply the reflectivity of the material at normal incidence [2]. Since $\theta_i = 0°$ in this case, the base reflectance can be computed as

$$F_0 = \left(\frac{\eta_i - \eta_t}{\eta_i + \eta_t}\right)^2, \tag{4}$$

since the full Fresnel equations simplify to Equation 4 at normal incidence. Once the Fresnel term (F_i) has been obtained, it is used to compute the final fragment colour as a weighted sum of the fragment colour found by sampling the environment using the refracted ray $\vec{T_1}$ ($Colour_{\vec{T_1}}$), and the colour found by sampling the environment using the reflected ray $\vec{R_1}$ ($Colour_{\vec{R_1}}$) using the equation

$$Colour_{Frag} = F_i Colour_{\vec{R_1}} + (1 - F_i) Colour_{\vec{T_1}}. \tag{5}$$

In other words, the Fresnel term determines how much the refracted ray colour and the reflected ray colour contribute to the final blended fragment colour. According to the law of reflection, the incident angle θ_i and the reflected angle θ_r are equal. The reflected ray R_1 is computed as

$$\vec{R_1} = \vec{V} - 2(\vec{V} \cdot \vec{N_1})\vec{N_1}, \tag{6}$$

which is simply the mirror of incident ray $\vec{V}$ across the surface by subtracting twice the projection of $\vec{V}$ onto $\vec{N_1}$ (assuming $\vec{N_1}$ is normalized to have a magnitude of one in this case). Once the reflected ray $\vec{R_1}$ has been found, it is used to sample the skybox cubemap to find a reflected colour for the fragment. The final fragment colour is then calculated using Equation 5.

Two-Surface Refraction

To implement refraction through two surfaces, the exit point ($\mathbf{P}_2$) of the refracted ray ($\vec{T_1}$) out the back of the 3D model must be found and the surface normal ($\vec{N_2}$) of the model at $\mathbf{P}_2$ must be retrieved. In a ray traced application, the path of the $\vec{T_1}$ is traced to the back of the model where the exact location of $\mathbf{P}_2$ and its corresponding surface normal $\vec{N_2}$ are found. The resulting refracted ray ($\vec{T_2}$) at the exit point ($\mathbf{P}_2$) is then computed and traced until the next intersection point (if it exists). This repeats until a either a maximum number of ray bounces have occurred or the ray reaches a light source in the scene. This is not possible in a rasterized implementation, which means we need another way to approximate $\mathbf{P}_2$ and $\vec{N_2}$.

Wyman's approach entails computing the distance from $\mathbf{P}_1$ to the back of the 3D model along the

incident ray ($\vec{V}$) as if the ray transmitted straight through the medium without bending. He calls this distance d_V in Figure 1 above. Additionally, he computes the distance from $\mathbf{P}_1$ to the opposite side of the 3D model along the inverted surface normal $-\vec{N_1}$. He calls this distance d_N in Figure 1. He then uses d_V and d_N to compute an approximation for the distance from $\mathbf{P}_1$ to $\mathbf{P}_2$, which he denotes as $\tilde{d}$. He computes $\tilde{d}$ as a weighted sum of d_V and d_N based on the ratio of the transmitted ray angle θ_t and the incident ray angle θ_i. As such, Wyman's approximation of $\tilde{d}$ is given by

$$\tilde{d} = \frac{\theta_t}{\theta_i} d_V + \left(1 - \frac{\theta_t}{\theta_i}\right) d_N. \tag{7}$$

By interpolating between d_V and d_N in this manner, more weight is assigned to d_V in the approximation of $\tilde{d}$ when the amount of refraction is small (θ_t is only a little smaller than θ_i). Conversely, if the ray refracts a lot (θ_t is much smaller than θ_i), then d_N is assigned more weight in the approximation of $\tilde{d}$. Having obtained an approximation for $\tilde{d}$, the location of the exit point is then approximated according to

$$\tilde{\mathbf{P}}_2 = \mathbf{P}_1 + \tilde{d}\vec{T_1} \approx \mathbf{P}_2 = \mathbf{P}_1 + d\vec{T_1}, \tag{8}$$

where d is the true distance from $\mathbf{P}_1$ to $\mathbf{P}_2$ and $\tilde{d}$ is the approximated distance between them. How the back-face surface normal $\vec{N_2}$ at $\mathbf{P}_2$ is found will be covered in the next section. Once $\vec{N_2}$ has been obtained, the refracted ray exiting the model ($\vec{T_2}$) can then be computed using the same method used to find $\vec{T_1}$ described above. One important change to make when computing $\vec{T_2}$ is to invert the ratio of the IORs since the ray is now exiting the transmission medium (the model) and entering the original incidence medium (the air). Once $\vec{T_2}$ has been obtained, it is used to sample the environment (i.e. the skybox cubemap) for the refracted colour of the fragment, which is then combined with the fragment colour of the reflected ray ($\vec{R_1}$) according to the value of the Fresnel term, as described in Equation 5.

Ray Tracing

The final part of my project entailed implementing an OpenGL ray traced method to serve as a visual quality ground truth against which to compare the one-surface and two-surface refraction approaches described above. The exact implementation of this will be detailed in the next section but some of the fundamentals will be detailed below. My ray traced implementation employs the Möller-Trumbore ray-triangle intersection algorithm described in [3]. The algorithm defines a fast and memory-efficient way of calculating whether or not a given ray intersects a specific triangle in the model mesh. A ray can be defined as

$$R(t) = O + t\vec{D}, \tag{9}$$

where t is the distance along the ray, $R(t)$ is the position along the ray at distance t, O is the origin of the ray, and D is the normalized direction vector of the ray. Let the three vertices of a triangle be denoted by V_0, V_1 and V_2. Any point $T(u, v)$ on a triangle is given by

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2, \tag{10}$$

where (u, v) are barycentric coordinates that enable any point on the surface of a triangle to be expressed as a weighted sum of its vertices. It is a requirement of barycentric coordinates that $u \geq 0$, $v \geq 0$, and $u + v \leq 1$ [3]. The algorithm begins by computing the two edge vectors that share V_0,

$$\vec{e_1} = V_1 - V_0 \text{ and } \vec{e_2} = V_2 - V_0, \tag{11}$$

before taking the cross product of $\vec{e_2}$ and the direction vector of the ray $\vec{D}$ to obtain a vector $\vec{p}$ perpendicular to both. The determinant is then computed as

$$det = \vec{e_1} \cdot \vec{p}, \tag{12}$$

as a means of checking if the ray is parallel to the plane of the triangle and therefore does not intersect with it (determinant very close to zero). If the determinant is non-zero, then the vector from vertex V_0 to the origin of the ray is computed as $\vec{t} = O - V_0$; and the cross product of $\vec{t}$ and $\vec{e_1}$ is calculated to obtain vector $\vec{q}$ perpendicular to both. The barycentric coordinates (u, v) are then computed as

$$u = \vec{t} \cdot \vec{p} \text{ and } v = \vec{D} \cdot \vec{q}. \tag{13}$$

If $u < 0$ or $u > det$, then the corresponding intersection was outside the bounds of the triangle (i.e. the ray did not hit the triangle). Similarly, if $v < 0$ or $u + v > det$, then the ray missed the triangle. If both barycentric coordinates are valid, then the distance along the ray t is computed as

$$t = \frac{\vec{e_t} \cdot \vec{q}}{det}. \tag{14}$$

If $t > 0$, then the hit is valid and the location of the hit is calculated as

$$P_{hit} = R(t) = O + t\vec{D}. \tag{15}$$

Implementation

This section details how I implemented each of the three methods (one-surface refraction, two-surface refraction, and ray tracing) in OpenGL C++. Note that in the previous section I have referred to the incident ray as the vector $\vec{V}$. However, from this point further I denote the incident ray (from the camera to the vertex) as $\vec{I}$ and the view ray (from the vertex to the camera) as $\vec{V}$. In other words, $\vec{V}$ and $\vec{I}$ are exact opposites of each other: $\vec{V} = -\vec{I}$.

One-Surface Refraction

One-surface refraction was implemented using a standard rasterization pipeline with GLSL vertex and fragment shaders. In the vertex shader, the surface normal $\vec{N}$ at each vertex is transformed into world space using the inverse transpose of the model matrix, while the view direction vector $\vec{V}$ (from the surface to the camera) is computed as the normalized vector from the world space fragment position to the camera position (derived from the inverse of the view matrix). A snippet of this implemented in the vertex shader is provided below. The entire vertex shader for one-surface refraction can be found in Appendix A at the end of the document.

```
// Convert vertex to world space
vec4 worldPos = model * vec4(aPos, 1.0);
vec3 camPos = vec3(inverse(view) * vec4(0.0, 0.0, 0.0, 1.0));

// View and normal in world space
V = normalize(camPos - worldPos.xyz);
N = normalize(mat3(transpose(inverse(model))) * aNormal);
```

In the fragment shader, the refracted ray direction $\vec{T}$ is computed using the GLSL *refract()* function, which applies Snell's Law (see Equation 2) based on the incident ray $\vec{I} = -\vec{V}$, the surface normal $\vec{N}$, and the ratio of the IORs (air to model). This refracted vector is then used to sample the environment from a cubemap skybox to obtain a refracted fragment colour, as shown in the fragment shader code snippet below.

```
// Compute refraction direction (air -> glass)
vec3 T1 = refract(V, N1, airIOR / modelIOR);
vec3 finalColour = texture(skybox, T1).rgb;
FragColor = vec4(finalColour, 1.0);
```

The ray that reflects back off the surface in direction $\vec{R}$ is computed with the GLSL *reflect()* function and is used to sample the environment from the cubemap skybox to obtain a reflected fragment colour. The refracted and reflected fragment colours are then blended using the Fresnel-Schlick approximation; as shown in the GLSL code snippet below.

```
*** Obtain T1 with refraction GLSL code from above ***

// Compute Fresnel term
```

```
float cosTheta = clamp(dot(V, -N1), 0.0, 1.0);
float fresnelTerm = fresnelSchlick(cosTheta);

// Compute reflection direction
vec3 R1 = reflect(V, N1);

// Sample skybox for reflection and refraction
vec3 reflectedColour = texture(skybox, R1).rgb;
vec3 refractedColour = texture(skybox, T1).rgb;

// Blend using Fresnel term
finalColour = mix(refractedColour, reflectedColour, fresnelTerm);
FragColor = vec4(finalColour, 1.0);
```

Two-Surface Refraction

To implement Wyman's method for two-surface refraction, the steps outlined above to compute refraction through the first surface (air to model) and the reflected ray are identical. At this point we know the following:

- The incident ray $\vec{I}$
- The IOR of air ($\eta_i = 1.0$) and the IOR of the model ($\eta_t \approx 1.5$ for glass)
- The vertex position $\mathbf{P}_1$ where $\vec{I}$ enters the medium
- The surface normal $\vec{N_1}$ at $\mathbf{P}_1$
- The refracted vector $\vec{T_1}$
- The reflected vector $\vec{R}$

Therefore, to implement two-surface refraction, we need to approximate the ray exit point $\mathbf{P}_2$ and the surface normal at this point $\vec{N_2}$. We can then use these values to compute the second refracted ray $\vec{T_2}$ exiting the medium and sample the skybox to obtain a refracted fragment colour. As mentioned in the *Background* section, this entails computing the distances d_V (the distance from $\mathbf{P}_1$ to the back of the model along $\vec{I}$) and d_N (the distance from $\mathbf{P}_1$ to the opposite side of the model along $-\vec{N_1}$). Once these values are obtained, the distance from $\mathbf{P}_1$ to $\mathbf{P}_2$ is approximated according to Equation 7; from which $\mathbf{P}_2$ is calculated using Equation 8.

Computing d_V

To calculate d_V I had to adopt a two-pass rendering approach in OpenGL. In the first pass, I culled the front faces of the model (using *glCullFace(GL_FRONT)*) and only rendered the back faces using back-face shaders. I then stored the back-face depth values in a *GL_DEPTH_COMPONENT* texture which I stored in a frame buffer object (FBO). As a side note, I also stored the back-face normals in a *GL_RGBA16F* texture in the same FBO, which I will elaborate on later when I explain finding the back-face normal $\vec{N_2}$. I then performed the second pass, which was a front-face rendering. For this step, I instead culled the back faces of the model (using *glCullFace(GL_BACK)*) and obtained the back-face depth values from the depth texture in my FBO; which I had set as a *uniform sampler2D* variable in my front-face fragment shader. The texture coordinates for the back-face texture uniforms were obtained by dividing the position of the current fragment by the size of the textures (which was the resolution of my screen in this project). Once these had been calculated, the back-face depth along the incident ray $\vec{I}$ was obtained from the depth texture, from which the point where $\vec{I}$ exits the model ($\mathbf{P}_V$) was found and converted into world coordinates. This allowed for d_V to be computed as the distance between the $\mathbf{P}_1$ and $\mathbf{P}_V$. The process described above was implemented in my front-faces fragment shader as follows.

```
// UV coords
vec2 uv = gl_FragCoord.xy / vec2(textureSize(backfaceDepthTex, 0));

// View-space depth from backface
float backDepth = texture(backfaceDepthTex, uv).r;
if (backDepth >= 0.999)
    discard;

vec3 P1 = FragPos;
vec3 PV = getWorldPosFromDepth(backDepth, uv);
d_V = length(PV - P1); // Convert depth to real-world view ray thickness
```

The function *getWorldPoseFromDepth()* is provided in Appendix C at the end of this document.

Pre-computing d_N

Computing d_N is an operation that is impractical to perform inside GLSL shaders. However, it is easy to pre-compute this value for each model using Blender and export the model with d_N stored as an attribute. As such, this was my approach for pre-computing d_N in this project. For each model, I imported it into Blender, created a new geometry nodes graph that did a raycast from each vertex along its inverted normal (i.e. inside the model) and stored the position of the first hit in a *Color Attribute* of the model named d_N. The full Blender geometry nodes graph used is shown in Figure 2 below.

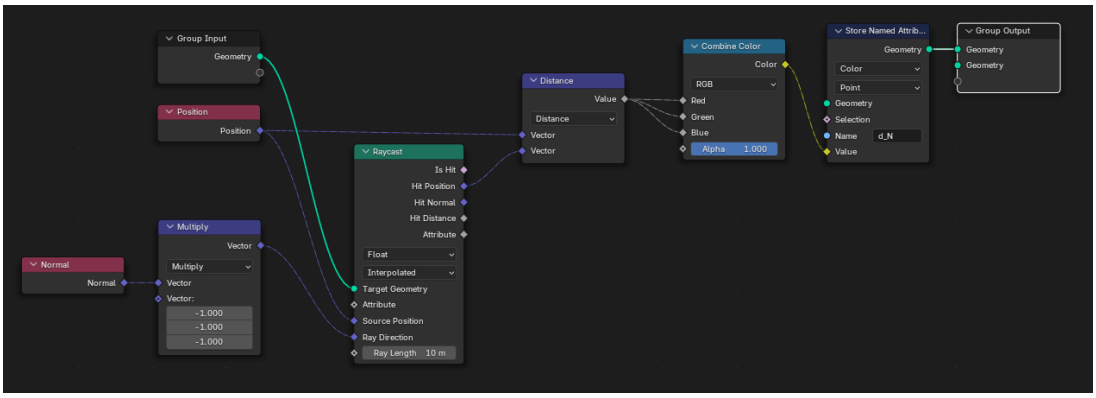

Figure 2: Precomputing d_N using a raycast node in Blender's geometry nodes

This approach works well for convex objects only. In the case of concave objects, however, the hit position of the raycast will not necessarily be the last exit point of the model. The raycast node stops as soon as a hit occurs and will store the location of the first hit as d_N, which is not the intended result. As such, I had to overcome this by implementing a Python script in Blender's scripting window. The logic of this Python script is identical to that of the geometry nodes raycast with the exception that it loops when a raycast hit occurs. As such, instead of breaking at the first hit and storing this position as d_N, I instead move the origin of the ray to the hit position, keep the direction (the inverted surface normal) the same, and do another raycast. This repeats until either a maximum number of raycast hits is reached or no hit occurs within the maximum ray length. This approach ensured that the true exit point ($\mathbf{P}_N$) along the inverted normal was found and the correct distance d_N from $\mathbf{P}_1$ to $\mathbf{P}_N$ was computed and stored as a color attribute of the model.

Finding $\vec{N_2}$

Once d_V and d_N had been obtained for each vertex position and used to approximate the distance d from $\mathbf{P}_1$ to $\mathbf{P}_2$; and hence find $\mathbf{P}_2$; the next step was to find the back-face surface normal at $\mathbf{P}_2$. As I mentioned earlier, during the back-face pass of my render loop, I stored the back-face depth values in a depth texture and the back-face surface normals in an RGBA textures. As such, once $\mathbf{P}_2$ had been obtained, I could project it into screen-space to obtain its texture coordinates in the back-face normals texture; from which I extracted $\vec{N_2}$. In my front-face fragment shader, this was implemented as follows.

```
// Compute approximation for d to find P2
float ratio = theta_t / theta_i;
float d = computeDistance(d_N, d_V, ratio);
vec3 P2 = P1 + T1 * d;

// Project P2 into screen space
vec4 clipP2 = projection * view * vec4(P2, 1.0);
clipP2 /= clipP2.w;
vec2 uvP2 = clipP2.xy * 0.5 + 0.5;

// Sample normal
vec3 N2 = texture(backfaceNormalTex, uvP2).rgb * 2.0 - 1.0;
```

In the code snippet above, the seemingly arbitrary scaling factors in the last two lines need to be addressed. The second last line calculates the texture coordinates of $\mathbf{P}_2$ (*uvP2*) as a function of the clip-space position of $\mathbf{P}_2$ (*clipP2*). Clip-space coordinates are expressed in normalized device coordinates (NDC) which fall in the range [-1, 1]. However, texture coordinates are in the range [0, 1]. As such, it is necessary to map the clip-space coordinates to texture coordinates by scaling them by 0.5 and adding 0.5. The last line finds the back-face surface normal $\vec{N}_2$ but indexing into the back-face normals texture at the computed texture coordinates. It then scales this value by 2.0 and subtracts 1.0. The reason for this is that the texture format in which the back-face normals are stored (*GL_ RGBA*) only takes on values in the range [0, 1]. Since surface normals take on values in the range [-1, 1], they need to be encoded by the back-face fragment shader, which entails mapping their values to the range [0, 1]. In my back-face shader, this was performed as follows.

```
// Encode the normals: map from [-1, 1] to [0, 1] for GL_RGBA compatibility
vec3 encodedNormal = normalize(worldNormal) * 0.5 + 0.5;
FragColor = vec4(encodedNormal, 1.0);
```

To undo this mapping in the front-face rendering pass, the normal has to be decoded, which entails doing the reverse mapping from [0, 1] to [-1, 1]. This reverse mapping is performed on the last line of the previous code snippet where $\vec{N}_2$ is decoded. Once $\vec{N}_2$ had been obtained, the refracted ray direction $\vec{T}_2$ out the back of the model was computed using the GLSL *refract()* method in the same manner it was used to find $\vec{T}_1$ through the first surface. This time it was called with $\vec{T}_1$ as the incident ray, $\vec{N}_2$ as the normal, and the ratio of IORs (model to air). $\vec{T}_2$ was then used to sample the environment to find the refracted ray colour from the skybox cubemap.

Raytracing in OpenGL

In this project I used a fragment shader to perform per-pixel ray traversal through the 3D scene with a maximum of four bounces inspired by the ray-traced approach proposed by Whitted in [4]. To begin, I stored the entire geometry (triangle vertices and their normals) of the 3D model in a Shader Storage Buffer Object (SSBO). Note that this requires the use of GLSL version 430 (not compatible with version 330). I rendered a full-screen triangle that was deliberately over-sized to fit the entire screen. At each pixel of the full-screen triangle I reconstructed a primary view ray by unprojecting the screen-space coordinates using the inverse of the projection and view matrices. Each ray was traced through the scene by looping over all triangles in the model and checking for intersections using the Möller-Trumbore ray-triangle intersection algorithm [3]. The closest triangle hit point was determined in this manner, and per-fragment normals were interpolated using barycentric coordinates. Refraction was simulated by alternating between the IORs of air and the model (glass in this project) at each bounce, and the ray was updated accordingly using the *refract()* function. Once the ray either missed the model geometry or reached the maximum of four bounces, the refracted and reflected fragment colours were obtained by sampling the environment. They were then mixed according to the Fresnel term. This process was very slow and was not practical for real-time applications. It was rather implemented to learn about ray tracing and to obtain a physically accurate representation of each model in a specific environment, against which I could compare my two refraction implementations.

Results

This section shows the results of the three implementations. The left-hand column shows one-surface refraction, the middle column shows two-surface refraction, and the right-hand column shows the ray-traced version. The evaluation and discussion of these results follows in the next section.

(a) (b) (c)

Figure 3: Sphere 3D model showing (a) refraction through one surface, (b) refraction through two-surfaces, (c) ray traced

(a) (b) (c)

Figure 4: Donut 3D model showing (a) refraction through one surface, (b) refraction through two-surfaces, (c) ray traced

(a) (b) (c)

Figure 5: Buddha 3D model showing (a) refraction through one surface, (b) refraction through two-surfaces using Chris Wyman's approach, (c) another refraction through two-surfaces approach

For each of the three implementations, the frames-per-second (FPS) were measured over a specified range of frames to ensure an accurate FPS measurement. For the one-surface and two-surface refraction approaches, the minimum, the maximum, and the average FPS were measured over 1000 frames. For the ray traced implementation (which was significantly slower), these measurements were performed over 100 frames. The laptop on which these tests were run has the following specification: AMD Ryzen 7 7840HS (8-core, 16-thread) CPU, NVIDIA RTX4060 GPU, 32GB RAM. The results of this experiment are tabulated below in Table 1.

Table 1: Minimum, maximum, and average frames per second for each method

		One Surface FPS			**Two Surfaces FPS**			**Ray Traced FPS**		
Model	**Polygons**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**
Monkey	967	153	556	819	108	375	449	9.78	10.75	11.91
Donut	4096	159	550	932	116	384	523	2.32	2.38	2.42
Sphere	8064	161	602	870	139	363	489	0.51	0.92	1.13
Buddha	9990	118	432	655	104	286	436	0.49	0.52	1.06
Teapot	15704	142	521	938	132	344	674	0.29	0.41	0.69

It is evident from Table 1 above that the minimum and maximum FPS measurements are suspiciously low/high in each instance, which I attribute to measurement artifacts in my FPS measurement class I implemented in this assignment. As such, the average FPS is the metric I will focus on to assess the performance of each method. To visualize these results, I plotted the average FPS achieved by each method on each model in Figure 6 below.

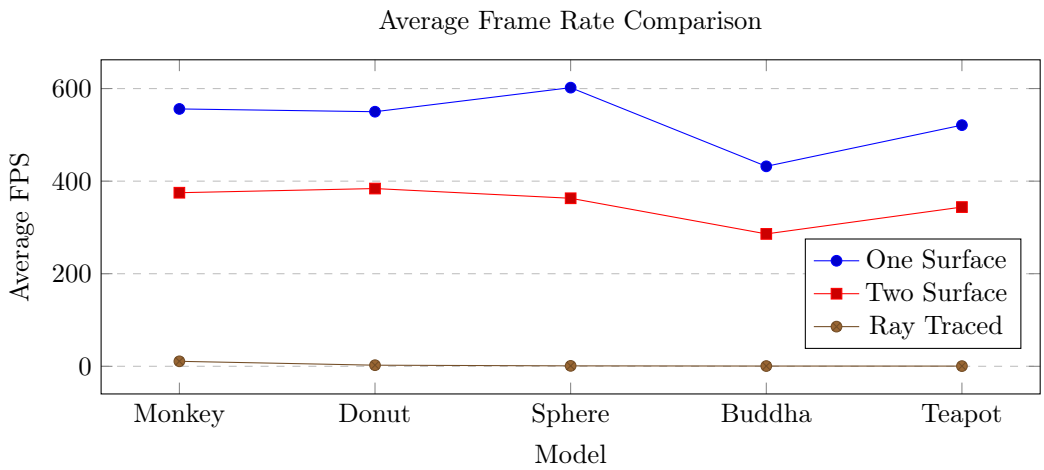

Figure 6: Average FPS comparison across different models and methods

Evaluation

FPS Results

The average FPS results provide some interesting insights into the performance of each method. It shows that the lower the polygon count of the model, the higher the average FPS achieved, which is an expected outcome. However, it is interesting to observe that this is not always the case. For instance, in almost all methods, the donut and sphere achieved a higher frame rate than the donut despite having much higher polygon counts. This can be attributed to the fact that vertex count isn't the sole factor that influences frame rate performance. Other more complex model attributes such as topology, triangle distribution, and size play an important role in FPS performance as well. This is most evident when considering the Buddha model which had by far the most complex topology (followed by the teapot, then the monkey). These models had the lowest average FPS performance when testing with the one-surface and two-surface refraction methods. In contrast to this, however, when measuring the FPS of the ray traced implementation, the performance decreased in an inversely proportional manner to the polygon count. In other words, for low polygon count the FPS was higher and for high polygon count the FPS was lower; without exception.

Regarding the performance of the one-surface refraction relative to the two-surface refraction, the one-surface method was consistently $\tilde{3}0$ - 40% faster than the two-surface approach. This was an expected outcome considering the two-surface method performed two rendering passes, had additional texture look-ups, and required additional computations to approximate the exit point $\mathbf{P}_2$. Despite the slower performance, the average FPS of the two-surface refraction method was still unbelievably high with a worst average performance of 286 FPS when rendering the Buddha model. This is an excellent result for real-time rendering applications.

Visual Quality Results

The two-surface approach achieved a significantly better visual result than the one-surface method across the board. In Figures 3, 4, and 5 above, it is evident that the two-surface refraction approach resembles the ray-traced ground-truth extremely closely. The one-surface approach is results in an unrealistic representation of refraction in that it refracts the ray only once. This is most evident in Sub-Figure 3a, where the sphere appears incredibly distorted relative to the other two methods.

Limitations and Improvements

The limitations of this project include the fact that the two-surface refraction proposed by Wyman is only accurate when rendering convex models and provides physically inaccurate results when rendering concave models due to the more complex path that light actually follows when refracting through these objects. Additionally, the project was limited to just simulating using the IOR of the model. It could be improved by adding additional material properties such as roughness, specular tint, light absorption/attenuation, chromatic dispersion, etc. It could also be improved by adding light sources to the scene instead of just sampling the skybox for the fragment colours. Finally, the ray-traced algorithm has much room for optimisation which was out of the scope of this project. For instance, a bounding volume hierarchy (BVH) or an alternative acceleration structure could be implemented to speed up the intersection tests and improve the real-time ray tracing performance.

Conclusion

In this project I sought to implement a two-surface refraction method proposed by Chris Wyman in his 2005 SIGGRAPH paper entitled *"An Approximate Image-Space Approach for Interactive Refraction"*. This entailed creating a two-pass rendering approach in which the back faces of the model were rendered first with depth and surface normal information stored in image textures, before rendering the front faces using depth and normal information from the back-face textures. In order to assess the performance of my implementation of Wyman's two-surface refraction method, I created a one-surface refraction approach and a ray-traced approach. The one-surface refraction approach achieved the best average frame rate performance out of the three method with a worst case result of 432 FPS. However, this approach had poor visual results that did not closely resemble the physically accurate ray-traced ground-truth render. The two-surface refraction method produced renders that appeared almost identical to the ray-traced version for convex models. This approach achieved an impressive worst case FPS performance of 286 FPS, with all other average FPS results over 300 FPS. This is an excellent real-time rendering result that achieves exceptionally realistic refraction through convex models. The gains in visual quality from the one-surface method to the two-surface method far outweigh the losses in frame rate.

YouTube Link

YouTube link: https://youtu.be/3ZKZjqSln1w

Declaration

I declare that all code and the writing of the report for this submission is entirely my own work and I have not collaborated with anyone or used any code found online.

Signed: Michael Millard
Date: 08/04/2025

Asset References

1. Suzanne the Monkey: https://sketchfab.com/3d-models/suzanne-blender-monkey-3a0647fba1654920b09ba4d11967e8cc
2. Utah Teapot: https://sketchfab.com/3d-models/the-utah-teapot-1092c2832df14099807f66c8b792374d
3. Teufelsberg Lookout Graffiti Skybox: https://polyhaven.com/a/teufelsberg_lookout
4. Rogland Clear Night Skybox: https://polyhaven.com/a/rogland_clear_night
5. Museum of Ethnography Skybox: https://polyhaven.com/a/museum_of_ethnography
6. Buddha: The 3D Buddha model used in this project was obtained from the demo code accompanying *"An Approximate Image-Space Approach for Interactive Refraction"* by Chris Wyman, available on his website https://cwyman.org/code/adaptiveCaustics_VS2008Code.zip. The Buddha model was in *.hem* format and was later converted to *.fbx* format by me. The original source of the model is not specified.
7. The sphere and donut were made in Blender.

References

[1] C. Wyman, "An approximate image-space approach for interactive refraction," in *Proceedings of SIGGRAPH 2005: Special Interest Group on Computer Graphics and Interactive Techniques*. ACM, July 2005. [Online]. Available: https://cwyman.org/papers/sig05_approxISRefr.pdf

[2] Z. Majercik, *The Schlick Fresnel Approximation*, 08 2021, pp. 109–114.

[3] T. Möller and B. Trumbore, "Fast, minimum storage ray-triangle intersection," *Journal of Graphics Tools*, vol. 2, 08 2005.

[4] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, p. 343–349, Jun. 1980. [Online]. Available: https://doi.org/10.1145/358876.358882

Appendix

A One-Surface Refraction Vertex Shader

```
#version 330 core

layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

out vec3 V; // View direction (from fragment to camera)
out vec3 N; // Normal vector

void main()
{
    vec4 worldPos = model * vec4(aPos, 1.0);
    vec3 camPos = vec3(inverse(view) * vec4(0.0, 0.0, 0.0, 1.0));

    // View and normal in world space
    V = normalize(camPos - worldPos.xyz);
    N = normalize(mat3(transpose(inverse(model))) * aNormal);

    gl_Position = projection * view * worldPos;
}
```

B One-Surface Refraction Fragment Shader

```
#version 330 core

in vec3 V; // View direction
in vec3 N; // Normal at the fragment

out vec4 FragColor;

// Skybox
uniform samplerCube skybox;

// Index of refratction
float airIOR = 1.0;
uniform float modelIOR;
uniform bool reflectEnable;

// Compute F0
float computeF0(float IOR_air, float IOR_surface)
{
```

```
    float ratio = (IOR_air - IOR_surface) / (IOR_air + IOR_surface);
    return ratio * ratio;
}

// Fresnel-Schlick approximation
float fresnelSchlick(float cosTheta)
{
    float F0 = computeF0(airIOR, modelIOR);
    return F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
}

void main()
{
    // Incident direction from eye to surface
    vec3 I = -V;

    // Compute refraction direction (air -> glass)
    vec3 refractedDir = refract(I, N, airIOR / modelIOR);
    vec3 finalColor = texture(skybox, refractedDir).rgb;

    // Check if reflection enabled
    if (reflectEnable)
    {
        // Compute Fresnel term
        float cosTheta = clamp(dot(I, -N), 0.0, 1.0);
        float fresnel = fresnelSchlick(cosTheta);

        // Compute reflection direction
        vec3 reflectedDir = reflect(I, N);

        // Sample skybox for reflection and refraction
        vec3 reflectedColor = texture(skybox, reflectedDir).rgb;
        vec3 refractedColor = texture(skybox, refractedDir).rgb;

        // Blend using Fresnel term
        finalColor = mix(refractedColor, reflectedColor, fresnel);
    }

    FragColor = vec4(finalColor, 1.0);
}
```

C Get World Position From Depth GLSL

```
// Convert screen-space depth to world-space position
vec3 getWorldPosFromDepth(float depth, vec2 uv)
{
    float z = depth * 2.0 - 1.0;
    vec4 clip = vec4(uv * 2.0 - 1.0, z, 1.0);
```

```
    vec4 viewPos = inverse(projection) * clip;
    viewPos /= viewPos.w;
    return vec3(inverse(view) * viewPos);
}
```

D Precomputing d_N in Blender Using a Python Script

```
import bpy
import mathutils
from mathutils import bvhtree, Vector
import os

# Settings
step_offset = 0.001   # Step forward after each hit
max_iterations = 10
max_distance = 100.0

# Get active mesh object
obj = bpy.context.object
if obj is None or obj.type != 'MESH':
    raise Exception("Please select a mesh object.")

# Apply transforms and set mesh
bpy.ops.object.transform_apply(location=True, rotation=True, scale=True)
mesh = obj.data

# Build BVH Tree
bvh = bvhtree.BVHTree.FromObject(obj, bpy.context.evaluated_depsgraph_get())

# Compute d_N for each vertex
d_n_values = [0.0 for _ in range(len(mesh.vertices))]
for i, vertex in enumerate(mesh.vertices):
    origin = obj.matrix_world @ vertex.co  # Vertex position in world space
    normal = vertex.normal.normalized()
    direction = -(obj.matrix_world.to_3x3() @ normal).normalized()
    current_origin = origin + direction * step_offset

    furthest_distance = 0.0
    remaining_distance = max_distance
    for _ in range(max_iterations):
        hit = bvh.ray_cast(current_origin, direction, remaining_distance)
        if hit[0] is None:
            break

        location, normal_hit, face_index, distance = hit
        if distance <= 0.0:
            break
```

```
        furthest_distance += distance
        current_origin = location + direction * step_offset
        remaining_distance = max_distance - furthest_distance
    d_n_values[i] = furthest_distance

# Create Color Point attribute named "d_N"
if "d_N" in mesh.attributes: # Remove if it already exists
    mesh.attributes.remove(mesh.attributes["d_N"])
color_attr = mesh.attributes.new(name="d_N", type='FLOAT_COLOR', domain='POINT')

# Assign RGB = d_N, A = 1.0
for i, val in enumerate(d_n_values):
    color_attr.data[i].color = (val, val, val, 1.0)
```