CPI 411 Graphics for Games

Monte-Carlo Path Tracing

This lab will serve as an introduction to the concepts of Monte-Carlo path tracing, working around the limitations of the Monogame framework and favoring code readability over performance.

A. First Concept

Introduced by James Kajiya in his paper on *The Rendering Equation*, path tracing was developed as an alternative to distributed ray tracing that sought to approximate a solution for the notorious equation. This version of the algorithm shoots a set of paths from the camera and through a viewport, with bounced ray directions being chosen probabilistically. This backwards process is done for efficiency, as all paths shot from the camera are guaranteed to contribute to the image.

B. Shader File

The path tracing algorithm will be developed within a single fragment shader. Create an effect file in MGCB titled 'path-tracer.fx'. Let's begin by listing all data structures used for the program, we will get into more detail about what these objects do later:

```
struct FS_Input {
    float4 Position : SV_POSITION;
    float4 Color : COLOR0;
    float2 TextureCoord : TEXCOORD0;
};
struct Ray {
    float3 Origin;
    float3 Direction;
};
struct Material {
    float4 Color;
    float3 lightColor;
    float  lightIntensity;
    float4 specularColor;
    float  specularIntensity; // AKA smoothness
    float  gloss; // probability of hit to be specular vs diffuse
};
struct Sphere {
    float3 Position;
    float radius;
    Material material;
};
struct Hit {
    bool hit;
    float distance;
    float3 Point;
    float3 Normal;
    Material material;
};
```

Optionally, create the following constructor functions to make our code easier to read later:

```
/* Ray Constructor */
Ray _ray(float3 origin, float3 target) {
    Ray r;
    r.Origin = origin;
    r.Direction = normalize(target - origin);
    return r;
}
/* Sphere Constructor */
Sphere newSphere(float3 position, float radius, Material material) {
    Sphere s;
    s.Position = position;
    s.radius = radius;
    s.material = material;
    return s;
}
/* Material Constructor */
Material newMaterial(float4 color, float3 lightColor, float lightIntensity,
float4 specularColor, float specularIntensity, float gloss) {
    Material m;
    m.Color = color;
    m.lightColor = lightColor;
    m.lightIntensity = lightIntensity;
    m.specularColor = specularColor;
    m.specularIntensity = specularIntensity;
    m.gloss = gloss;
    return m;
}
```

Our first issue to tackle is determining the point in our viewport that we must trace a ray through. If a ray's origin is the camera position and its direction is determined by a point at the viewport, we must convert texture-space to camera-space coordinates. Chapter 8 of *The Cg Tutorial* (NVIDIA, 2007) details the following function to expand a range-compressed vector:

```
float3 expand(float3 v) { return (v - 0.5) * 2; } // from [0, 1] to [-1,1]
```

We can then use this function on the current UV values, and scale the result according to the dimensions of our digital viewport. Our camera is assumed to be oriented facing the Z-axis, so the distance between the camera and viewport will be Z in the resulting vector.

```
/* Calculates Local Position of a Point in the Viewport [Assumes camera faces +Z]*/
float3 ViewportPointLocal(float2 uv, float focalLength, float2 viewportSize) {
    float3 local = expand(float3(uv, 1));
    return float3(-(local.x * viewportSize.x), -(local.y * viewportSize.y),
focalLength);
}
```

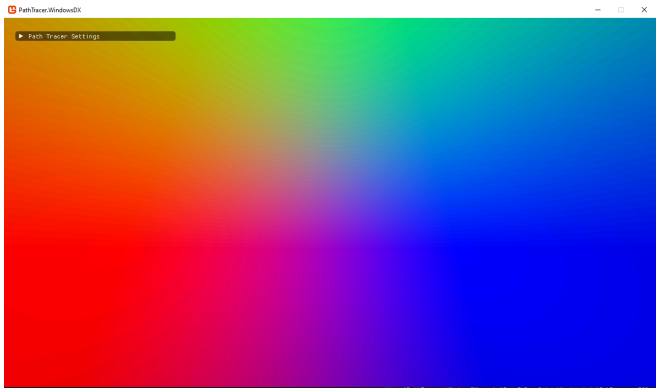
Take note of the negation applied to both X and Y values of the resulting vector; Monogame uses a right-handed coordinate system, so both values must be flipped to account for the UV

coordinates which increase along flipped axes.

With our set up complete, we can begin work on our fragment shader. Create the following variables and write a simple fragment shader function that will be changed later.

```
float3 _cameraPosition; // Camera Position in World-Space Coordinates
float4x4 _cameraTransform; // Camera Transformation Matrix
float _focalLength; // d: Length between Camera & Viewport
float2 _viewportSize;

sampler TextureSampler : register(s0) { // s0 is targeted by SpriteBatch on Draw
    Texture = <PrevRender>;
};

float4 FS_Main(FS_Input input) : COLOR0 {
    float4 color = tex2D(TextureSampler, input.TextureCoord);

    /* pixel coordinates as a percentage of the screen, from 0 to 1 on each axis */
    float2 uv = input.TextureCoord.xy;

    /* Calculate the Position for a given Point in the Viewport */
    float3 localPoint = ViewportPointLocal(uv, _focalLength, _viewportSize);
    float3 Point = mul(float4(localPoint, 1), _cameraTransform).xyz;

    /* Cast Ray from camera position towards Viewport Point */
    Ray r = _ray(_cameraPosition, Point);

    return float4(r.Direction, color.w);
}
```

Take a moment to look at the calculation of "Point" in our shader. Recall the viewport point is positioned within camera-space coordinates, that is, being oriented along the position of the camera. We use the camera's transformation matrix to convert this point to world-space coordinates.



Check your work, it should look something like the image to the right. Notice how ray direction corresponds to RGB values.

In order to introduce randomness within our shader, we need to implement a function that can generate random values from the GPU. In his article *Building a Fast, SIMD/GPU-friendly Random Number Generator For Fun And Profit*, Parker introduces a technique for random float generation that involves a combination of the LCG and XOR-shift algorithms.

```
uint lcg_xs_24(inout uint state) {
  uint result = state * 747796405u + 2891336453u;
  uint hashed_result = result ^ (result >> 14);
  state = hashed_result;
  return hashed_result >> 8;
}

float rand(inout uint state) {
  uint result = lcg_xs_24(state);
  const float inv_max_int = 1.0 / 16777216.0;
  return float(result) * inv_max_int;
}
```

We must then transform these uniform random values into a spherically symmetric normal distribution. For this we use the Box-Muller transform in basic cartesian form, developed in 1958 by George Box and Mervin Muller.

```
/* Converts uniformly distributed random float to gaussian/normal distribution */
float gauss(inout uint state, float mean = 0, float sd = 1) {
    float theta = 2 * 3.1415926 * rand(state);
    float rho = sqrt(-2 * log(rand(state)));
    return mean + (sd * rho) * cos(theta);
}

float3 Random(inout uint state) {
    float3 dir = float3(gauss(state), gauss(state), gauss(state));
    return normalize(dir);
}
```

This ‘gauss’ function ensures that our rays will bounce uniformly around the hemisphere of a surface normal. In reality, rays that are less aligned with the surface normal tend to contribute less light to the surface. This will be addressed later with cosine weighting.

Before we begin path tracing, let’s address a speed bump that comes with using the monogame framework. Monogame lacks 'StructuredBuffer' data types that would allow us to send sphere data to the GPU in sequential order. Instead, we work around this limitation by sending sphere information via multiple arrays, each containing the values of one specific sphere parameter. Add the following variables to ‘path-tracer.fx’.

```
float3 SPHERE_POS[NUM_SPHERES];
float  SPHERE_RADIUS[NUM_SPHERES];
float4 SPHERE_DIFF_COL[NUM_SPHERES];
float4 SPHERE_SPEC_COL[NUM_SPHERES];
float3 SPHERE_LITE_COL[NUM_SPHERES];
float  SPHERE_SPEC[NUM_SPHERES];
float  SPHERE_GLOSS[NUM_SPHERES];
float  SPHERE_LITE[NUM_SPHERES];
```

Next, add the following function to look for intersections of a given ray against each sphere in our scene. The function keeps track of the intersection closest to the ray's origin, and returns it as the result after iterating through every sphere.

```
/* CalculateSpheres(R) */
Hit CalculateSpheres(Ray R) {
    /* We haven't hit anything, so our closest hit is infinitely far away */
    Hit closestHit = (Hit) 0;
    closestHit.distance = 1000000000;

    for (int i = 0; i < NUM_SPHERES; i++) {
        Material mat = newMaterial(SPHERE_DIFF_COL[i], SPHERE_LITE_COL[i],
SPHERE_LITE[i], SPHERE_SPEC_COL[i], SPHERE_SPEC[i], SPHERE_GLOSS[i]);
        Sphere sphere = newSphere(SPHERE_POS[i], SPHERE_RADIUS[i], mat);

        Hit hit = IntersectSphere(R, sphere);

        if (hit.hit && hit.distance < closestHit.distance) {
            closestHit = hit;
            closestHit.material = sphere.material;
        }
    }

    return closestHit;
}
```

We now need a method to perform the hit calculation of a given ray and sphere. We can achieve this by solving a quadratic formula. This code is based on the pseudo-code instructions discussed by Gabriel Gambetta in his book *Computer Graphics from Scratch.* (Chapter 2: Basic Raytracing, page 25)

```
float3 RayAt(Ray R, float t) {
    return R.Origin + (t * R.Direction);
}

Hit IntersectSphere(Ray R, Sphere S) {
    Hit hitInfo = (Hit) 0;

    /* CO - offset between the Ray's Origin & Sphere's Center */
    float3 CO = R.Origin - S.Position;

    /* Calculating quadratic equation variables */
    float a = dot(R.Direction, R.Direction);
    float b = 2 * dot(CO, R.Direction);
    float c = dot(CO, CO) - pow(S.radius, 2);

    float discriminant = b * b - 4 * a * c;

    /* The Ray Missed if there is no solution (discriminant < 0) */
    if (discriminant >= 0) {
        /* Calculate distance to the nearest Intersection between Ray and Sphere */
        float dist = (-b - sqrt(discriminant)) / (2 * a);
```

```
        /* Ignore the Hit if it occurs behind the camera */
        if (dist < 0) return hitInfo;

        hitInfo.hit = true;
        hitInfo.distance = dist;
        hitInfo.Point = RayAt(R, dist);
        hitInfo.Normal = normalize(hitInfo.Point - S.Position);
    }

    return hitInfo;
}
```

Finally, the Trace function simulates light hitting a surface over a maximum number of bounces. This incoming light is tinted by one of the material colors of the intersected sphere, with the probability of the color being specular corresponding to the material's glossiness. The direction of the next bounce is also dependent on the material; its trajectory is linearly interpolated between a random diffuse direction and a specular reflection. This is, in effect, Path Tracing's approximation to solving the rendering equation.

```
float3 Trace(Ray r, inout uint rng) {
    /* Keeping track of light and color */
    float3 incomingLight = float3(0, 0, 0);
    float3 rayColor = float3(1, 1, 1);

    for (int i = 0; i <= BOUNCES; i++) {
        Hit hit = CalculateSpheres(r);
        Material mat = hit.material;
        if (!hit.hit) {
            incomingLight += float3(0.0, 0.0, 0.0) * rayColor;
            break;
        }
        /* Use diffuse or specular based on material's gloss */
        bool isReflected = mat.gloss >= rand(rng);
        /* Calculate light from traced ray hit */
        float3 lightEmmitted = mat.lightColor * mat.lightIntensity;
        /* Add to hit info to our values */
        incomingLight += lightEmmitted * rayColor; // if ray hits light source
        rayColor *= lerp(mat.Color, mat.specularColor, isReflected).xyz;
        /* Randomize the direction of the next bounce. */
        float3 dirDiffuse  = normalize(hit.Normal + Random(rng)); // cosine-weighted
        float3 dirSpecular = reflect(r.Direction, hit.Normal);
        r.Origin = hit.Point;
        r.Direction = normalize(
            lerp(dirDiffuse, dirSpecular, hit.material.specularIntensity * isReflected)
        );
    }

    return incomingLight;
}
```

With this we can update our fragment shader to render a scene.

```
int SPP = 5; // (Samples-Per-Pixel) # of rays traced per pixel
int BOUNCES = 10; // Maximum # of bounces to calculate on a ray
int _frame; // Index of current frame; used to randomize
int _accumulated; // # of accumulated frames
bool _accumWeighted;
int _screenX, _screenY; // Width & Height of screen (in pixels)

float4 FS_Main(FS_Input input) : COLOR0 {
    float4 color = tex2D(TextureSampler, input.TextureCoord);
    /* pixel coordinates as a percentage of the screen, from 0 to 1 on each axis */
    float2 uv = input.TextureCoord.xy;
    /* Calculate the Position for a given Point in the Viewport */
    float3 localPoint = ViewportPointLocal(uv, _focalLength, _viewportSize);
    float3 Point = mul(float4(localPoint, 1), _cameraTransform).xyz;
    /* Cast Ray from camera position towards Viewport Point */
    Ray r = _ray(_cameraPosition, Point);
    /* Create RNG Seed */
    uint2 pixelCoord = uv * uint2(_screenX, _screenY);
    uint pixelIdx = pixelCoord.y * _screenX + pixelCoord.x;
    uint rng = (pixelIdx + _frame * 2654435761u + 1692572869u);
    rng = rng ^ (rng >> 18);
    lcg_xs_24(rng); // introduce more randomness with additional iteration of LCG-XS
    // Trace Pixel Color
    float3 totalPixelColor = float3(0, 0, 0);
    for (int rayIdx = 0; rayIdx < SPP; rayIdx++) {
        totalPixelColor += Trace(r, rng);
    }
    float3 avgColor = totalPixelColor / SPP;
    /* Return the accumulated color */
    return Accumulate(float4(avgColor, 1), color, _accumulated, _accumWeighted);
}
```

As a final touch, set up a simple temporal accumulation filter to average each render with the previous frame. This can be achieved via a simple lerp function:

$$C'i = a \times Ci + (1 - a) \times C'i - 1$$

```
float4 Accumulate(float4 colorCurr, float4 colorPrev, int accumulated, bool weighted) {
    if (accumulated == 0) return colorCurr;
    float a = (weighted) ? (1.0 / (accumulated + 1)) : 0.2f;
    return saturate(lerp(colorCurr, colorPrev, 1 - a));
}
```

This function allows for two different ratios of a: weighted and constant. A constant ratio will calculate the resulting color with a ratio of 1:5. A weighted average will instead favor the earlier frames of an image, according to the total number of frames accumulated. As we'll soon see, this results in marked differences in image quality.

C. Main Program (Game_PathTraccer.cs)

Start by setting up the game class properties.

```
private GraphicsDeviceManager _graphics;
private SpriteBatch _spriteBatch;
private (int Width, int Height, float AspectRatio) _screen;

private RenderTarget2D _pathTarget;
private RenderTarget2D _rasterTarget;
private Texture2D _renderPrev;
private Color[] _textureData;

private Effect _effect;
private SpriteFont _font;
private Model _sphere;
private PathTraceCamera _camera;

/* CONTROL HANDLER: Polls for mouse movement and updates camera transform accordingly */
private ControlHandler _controlHandler = new();
MouseDrag _cameraRot = new(new(0.5f, 0.7f), MouseButtonType.RIGHT, value: new(0, 0));
MouseDrag _cameraXY =  new(new(0.05f, 0.07f), MouseButtonType.RIGHT, value: new(0, 0));
MouseDrag _cameraZ =   new(new(0.05f, 0.07f), MouseButtonType.WHEEL, value: new(0, -5));

/* SCENE DATA */
ModelList _spheres = new();

/* GUI COMPONENTS */
private MainWindow _GUI;
private SceneWindow _sceneEdit;
private ProfilerWindow _profiler;
private SettingsWindow _settings;
```

Set the graphics profile to ‘HiDef’ inside the game constructor. Optionally, you can set ‘IsFixedTimeStep’ to false in order to remove the framerate cap of the draw call. Since we are not doing any animations, this should not cause any issues.

```
public Game_PathTracer() {
    _graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    this.IsMouseVisible = true;
    this.IsFixedTimeStep = false;

    _graphics.GraphicsProfile = GraphicsProfile.HiDef;
}
```

Set up some properties of the game inside the initialize method. Notice the two render targets we will use to render both rasterized and path traced visuals. Also, notice ‘_renderPrev’ and ‘_textureData’. These objects will keep track of the previous frame’s color information, which will be passed to the shader for use within the accumulation function.

```
protected override void Initialize() {
    /* Request preferred resolution & keep track of screen information */
    _graphics.PreferredBackBufferWidth = 1280;
    _graphics.PreferredBackBufferHeight = 720;
    _graphics.ApplyChanges();
    _screen = (
        GraphicsDevice.Viewport.Width,
        GraphicsDevice.Viewport.Height,
        GraphicsDevice.Viewport.AspectRatio
    );
    /* Initialize GUI Windows */
    _GUI = new(this, GraphicsDevice);
    _sceneEdit = new();
    _settings = new();
    _profiler = new();
    /* Main Camera */
    _camera = new PathTraceCamera(
        position: new(_cameraXY.Value, _cameraZ.Y),
        lookAt: Vector3.Zero,
        perspective: new Projection(60, _screen.AspectRatio, (0.1f, 100))
    );
    /* Render Target to Draw (Path Tracing) */
    _pathTarget = new RenderTarget2D(
        graphicsDevice: GraphicsDevice,
        width: _screen.Width,
        height: _screen.Height,
        mipMap: false,
        preferredFormat: GraphicsDevice.PresentationParameters.BackBufferFormat,
        preferredDepthFormat: DepthFormat.Depth24Stencil8,
        preferredMultiSampleCount: GraphicsDevice.PresentationParameters.MultiSampleCount,
        usage: RenderTargetUsage.PreserveContents
    );
    /* Render Target to Draw (Rasterization) */
    _rasterTarget = new RenderTarget2D(
        graphicsDevice: GraphicsDevice,
        width: _screen.Width,
        height: _screen.Height,
        mipMap: false,
        preferredFormat: GraphicsDevice.PresentationParameters.BackBufferFormat,
        preferredDepthFormat: DepthFormat.Depth24Stencil8,
        preferredMultiSampleCount: GraphicsDevice.PresentationParameters.MultiSampleCount,
        usage: RenderTargetUsage.DiscardContents
    );
    /* Instantiate texture that will hold the previous renders */
    _renderPrev = new Texture2D(GraphicsDevice, _screen.Width, _screen.Height);
    /* Instantiate array that will be used to copy render data between GPU & CPU */
    _textureData = new Color[_pathTarget.Width * _pathTarget.Height];
    base.Initialize();
}
```

Load the effect, sphere model, and font information for the GUI inside of LoadContent:

```
protected override void LoadContent() {
        _spriteBatch = new SpriteBatch(GraphicsDevice);

        _GUI.onLoad();
        _font = Content.Load<SpriteFont>("fonts/font");
        _sphere = Content.Load<Model>("models/sphere/sphere");
        _effect = Content.Load<Effect>("shaders/path-tracer");
        _spheres.onLoad(_effect);
}
```

The update function simply polls mouse drags in order to move the camera, as the rest is handled by the GUI using ImGuiNet Monogame.

```
protected override void Update(GameTime gameTime) {
        _controlHandler.MouseCurrent(Mouse.GetState());
        _controlHandler.KeyboardCurrent(Keyboard.GetState());

        if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed
            || Keyboard.GetState().IsKeyDown(Keys.Escape))
            Exit();

        /* Poll Camera Transformation */
        _controlHandler.PollMouseDrag(_cameraRot, (Vector2 v) => _camera.Rotate(new(v, 0)));
        _controlHandler.PollMouseDrag(_cameraZ, (Vector2 v) => _camera.Translate(
             new(_camera.Offset.M41, _camera.Offset.M42, v.Y)
        ));

        _controlHandler.MousePrevious(Mouse.GetState());
        _controlHandler.KeyboardPrevious(Keyboard.GetState());
        base.Update(gameTime);
}
```

Set up an optional rasterization pass in order to compare our path traced scene with an equivalent rasterized image.

```
private void RasterPass(RenderTarget2D target) {
        GraphicsDevice.SetRenderTarget(target);
        GraphicsDevice.Clear(new Color(0, 0, 0));
        GraphicsDevice.BlendState = BlendState.Opaque; // Needed for DrawString() function
        GraphicsDevice.DepthStencilState = new DepthStencilState(){DepthBufferEnable = true};
        GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;
        foreach (Sphere s in _spheres.Spheres) {
                DrawSphere(s);
        }
        /* Drop the render target */
        GraphicsDevice.SetRenderTarget(null);
}
```

```
private void DrawSphere(Sphere s) {
    Vector3 color = new(s.DiffuseColor.X, s.DiffuseColor.Y, s.DiffuseColor.Z);
    Vector3 specular = new(s.SpecularColor.X, s.SpecularColor.Y, s.SpecularColor.Z);
    Matrix transform = (Matrix.CreateScale(s.Radius)*Matrix.CreateTranslation(s.Position));
    foreach (ModelMesh mesh in _sphere.Meshes) {
       foreach (ModelMeshPart part in mesh.MeshParts) {
              BasicEffect effect = (BasicEffect)part.Effect;
              effect.EnableDefaultLighting();
              effect.PreferPerPixelLighting = true;
              effect.DiffuseColor = color;
              effect.EmissiveColor = s.LightColor;
              effect.World = mesh.ParentBone.Transform * transform;
              effect.View = _camera.View;
              effect.Projection = Matrix.CreatePerspectiveFieldOfView(
              fieldOfView: MathHelper.ToRadians(98.3f), aspectRatio: 16/9.02f, 0.1f, 100);
       }
       mesh.Draw();
    }
}
```

Next, set up a path traced pass using a spritebatch.

```
private void PathTracePass(RenderTarget2D target) {
   GraphicsDevice.SetRenderTarget(target); // Set the Render Target & Clear it
   GraphicsDevice.Clear(ClearOptions.Target|ClearOptions.DepthBuffer, Color.Blue, 1.0f, 0);
   _effect.Parameters["_cameraPosition"].SetValue(_camera.Position);
   _effect.Parameters["_cameraTransform"].SetValue(
      Matrix.Multiply(_camera.Offset, Matrix.CreateFromQuaternion(_camera.Rotation))
   );
   _effect.Parameters["_viewportSize"].SetValue(_camera.ViewportDimensions);
   _effect.Parameters["_focalLength"].SetValue(_camera.Clip.Near);
   _effect.Parameters["_frame"].SetValue((int)_profiler.TotalFrames);
   _effect.Parameters["_screenX"].SetValue(_screen.Width);
   _effect.Parameters["_screenY"].SetValue(_screen.Height);
   _effect.Parameters["SPP"].SetValue(_settings.Samples);
   _effect.Parameters["BOUNCES"].SetValue(_settings.Bounces);
   _effect.Parameters["_accumulated"].SetValue((int)_settings.AccumulatedFrames);
   _effect.Parameters["_accumWeighted"].SetValue(
       _settings.Accumulation == Accumulate.WEIGHTED
   );
   _spriteBatch.Begin( // Draw Spritebatch with Path Tracing Effect
       SpriteSortMode.Deferred, BlendState.Opaque,
       rasterizerState: RasterizerState.CullNone, effect: _effect
   );
   _spriteBatch.Draw(
       _renderPrev,
       new Rectangle(0, 0, _screen.Width, _screen.Height),
       Color.White
   );
    _spriteBatch.End();
    GraphicsDevice.SetRenderTarget(null); // Drop the render target
}
```

Finally, our Draw function. Notice the calls for ‘SetData’ and ‘GetData’ before and after our path trace pass. The data array will be refreshed at each draw call in order to accumulate our results. Keep in mind, these are incredibly expensive operations and should be used sparingly.

```
protected override void Draw(GameTime gameTime) {
	/* Store previous render data in _RenderPrev [WARNING: expensive operation] */
	_renderPrev.SetData(_textureData);
	/* Rasterized Pass */
	RasterPass(_rasterTarget);
	/* Path Trace Pass */
	PathTracePass(_pathTarget);
	/* Retrieve the current render data from GPU [WARNING: expensive operation] */
	_pathTarget.GetData(_textureData);
	/* Draw both renders to screen */
	_spriteBatch.Begin(
		SpriteSortMode.Deferred, BlendState.AlphaBlend,
		rasterizerState: RasterizerState.CullNone, effect: null
	);
	_spriteBatch.Draw(
		(_settings.View) ? _pathTarget : _rasterTarget,
		new Rectangle(0, 0, _screen.Width, _screen.Height),
		Color.White
	);
	_spriteBatch.Draw(
		(_settings.View) ? _rasterTarget : _pathTarget,
		new Rectangle(
			x: GraphicsDevice.Viewport.Bounds.Right  - (_screen.Width / 4) - 25,
			y: GraphicsDevice.Viewport.Bounds.Bottom - (_screen.Height / 4) - 25,
			width:  _screen.Width / 4,
			height: _screen.Height / 4
		),
		Color.White
	);
	_spriteBatch.End();
	base.Draw(gameTime);
	/* Draw GUI & All Sub Windows */
	_GUI.BeginDraw(gameTime);
	_sceneEdit.DrawGUI("Scene Settings", _spheres, _effect);
	_settings.DrawGUI("Render Settings");
	_profiler.DrawGUI("Profiling", gameTime);
	_GUI.EndDraw();
}
```

Set up a scene within the ModelList class. Be sure to define the following constant inside `path-tracer.fx` equal to the number of objects present in the scene.

```
#define NUM_SPHERES 10
```

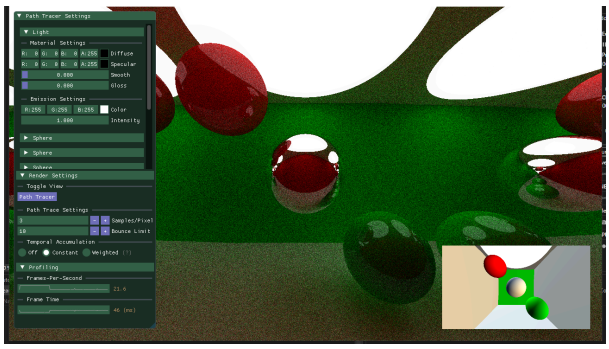
D. Main Exercise

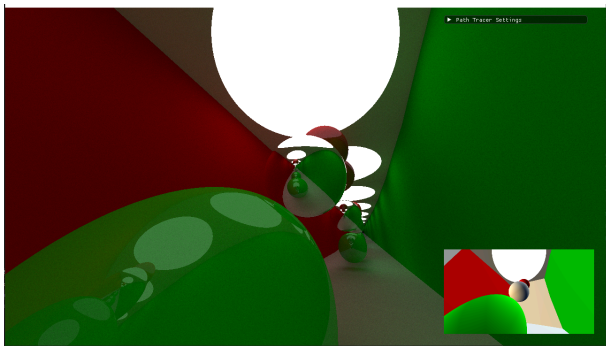
So far, we have been able to calculate ray intersections for spheres. However, in order to extend the program to render any kind of model, we must develop the ability to calculate ray intersections for triangles. Refactor the shader code to include the Möller–Trumbore ray-triangle intersection algorithm. This code was originally written by Sebastian Lague in his series *Coding Adventures: Ray Tracing,* and adapted to work inside our shader.

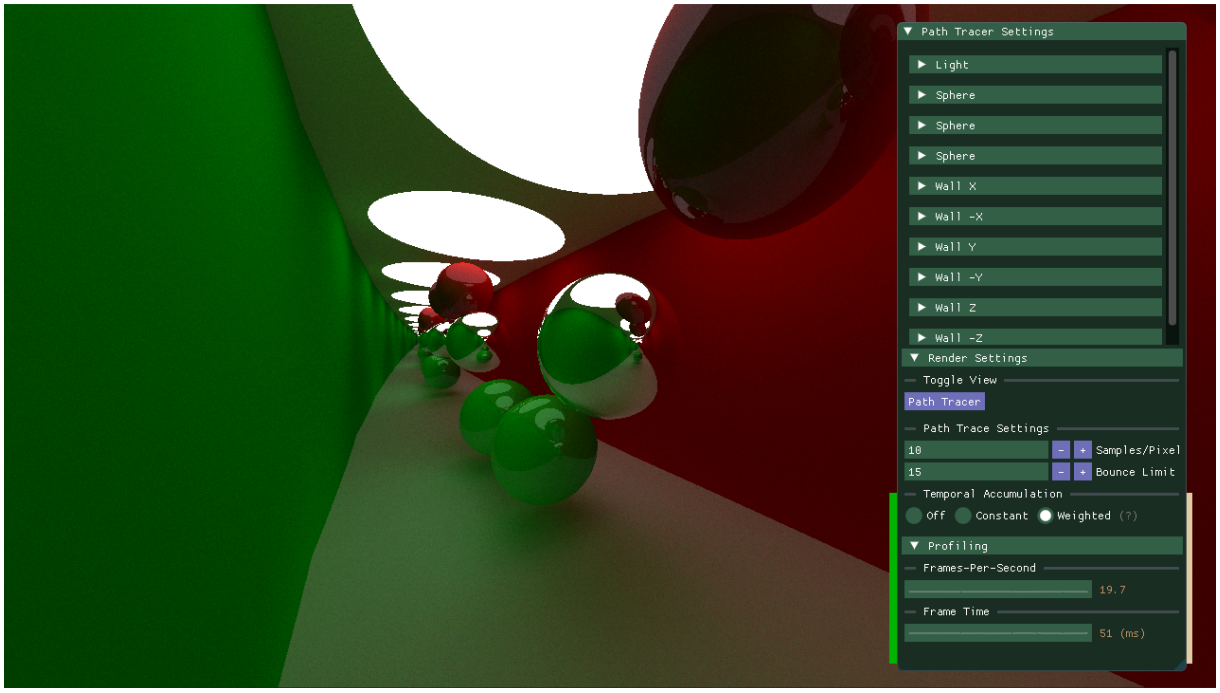
```
HitInfo IntersectTriangle(Ray r, Triangle tri) {
	// Initialize hit info
	Hit hit = (Hit) 0;
	// Calculate the edges A-B and A-C
	float3 AB = tri.PositionB - tri.PositionA;
	float3 AC = tri.PositionC - tri.PositionA;
	// Calculate the triangles normal
	float3 triNormal = cross(AB, AC);
	// Calculate the determinant
	float3 ao = r.Origin - tri.PositionA;
	float3 dao = cross(ao, r.Direction);
	float det = -dot(r.Direction, triNormal);
	float invDet = 1 / det;
	// Calculate the distance between the ray & triangle
	float d = dot(ao, triNormal) * invDet;
	// Calculate barycentric coordinates at intersection
	float u = dot(AC, dao) * invDet;
	float v = -dot(AB, dao) * invDet;
	float w = 1 - u - v;
	// Create hit information
	hit.hit = det >= 1E-6 && d >= 0 && u>=0 && v>=0 && w>=0;
	hit.Point = r.Origin + r.Direction * d;
	hit.Normal = normalize(tri.NormalA * w + tri.NormalB * u + tri.NormalC * v);
	hitInfo.distance = d;

	return hitInfo;
}
```

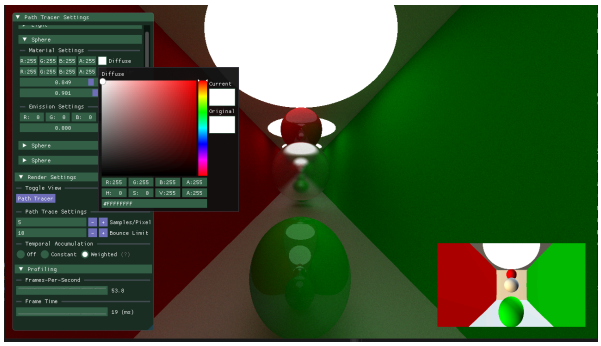
***** IMPORTANT *****

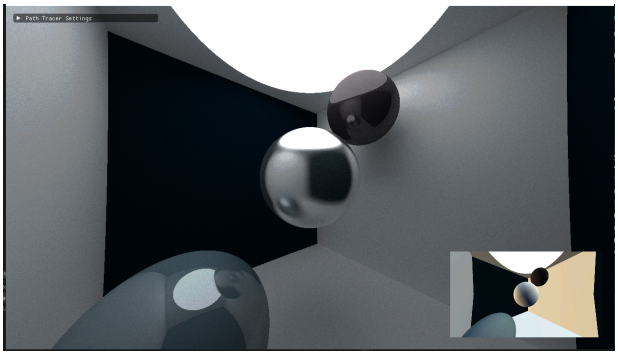
Complete the exercise in D section, and submit a zipped file including the solution (.sln) file and the project folders to the course's online site. The submission item is located in the "**Quiz and Lab**" section. Each lab has **10 points**. If you complete the exercise in class time, the full points will be assigned. The late submission is accepted just before the next class with 2 points reductions, because the solution is demonstrated in the next class.






Path Tracer Settings
Light
Sphere
Sphere
Sphere
Wall X
Wall -X
Wall Y
Wall -Y
Wall Z
Wall -Z
Render Settings
Toggle View
Path Tracer
Path Trace Settings
10
Samples/Pixel
15
Bounce Limit
Temporal Accumulation
Off
Constant
Weighted (?)
Profiling
Frames-Per-Second
19.7
Frame Time
51 (ms)





References

Caden Parker LCG-XS. 2015. *Building a Fast, SIMD/GPU-friendly Random Number Generator For Fun And Profit* https://vectrx.substack.com/p/lcg-xs-fast-gpu-rng

Gabriel Gambetta. 2021 *Computer Graphics from Scratch: A Programmer's Introduction to 3D Rendering.*

G. E. P. Box, Mervin E. Muller "*A Note on the Generation of Random Normal Deviates*," The Annals of Mathematical Statistics, Ann. Math. Statist. 29(2), 610-611, (June, 1958) [ex]

James T. Kajiya. 1986. *The Rendering Equation.* SIGGRAPH Comput. Graph. 20, 4 (Aug. 1986), 143–150. https://doi.org/10.1145/15886.15902

Pugeault, N., & Krüger, N. (2011). *Temporal accumulation of oriented visual features*. Journal of Visual Communication and Image Representation, https://doi.org/10.1016/j.jvcir.2010.12.001

Randima Fernando and Mark J. Kilgard. 2003. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Longman Publishing Co., Inc., USA.