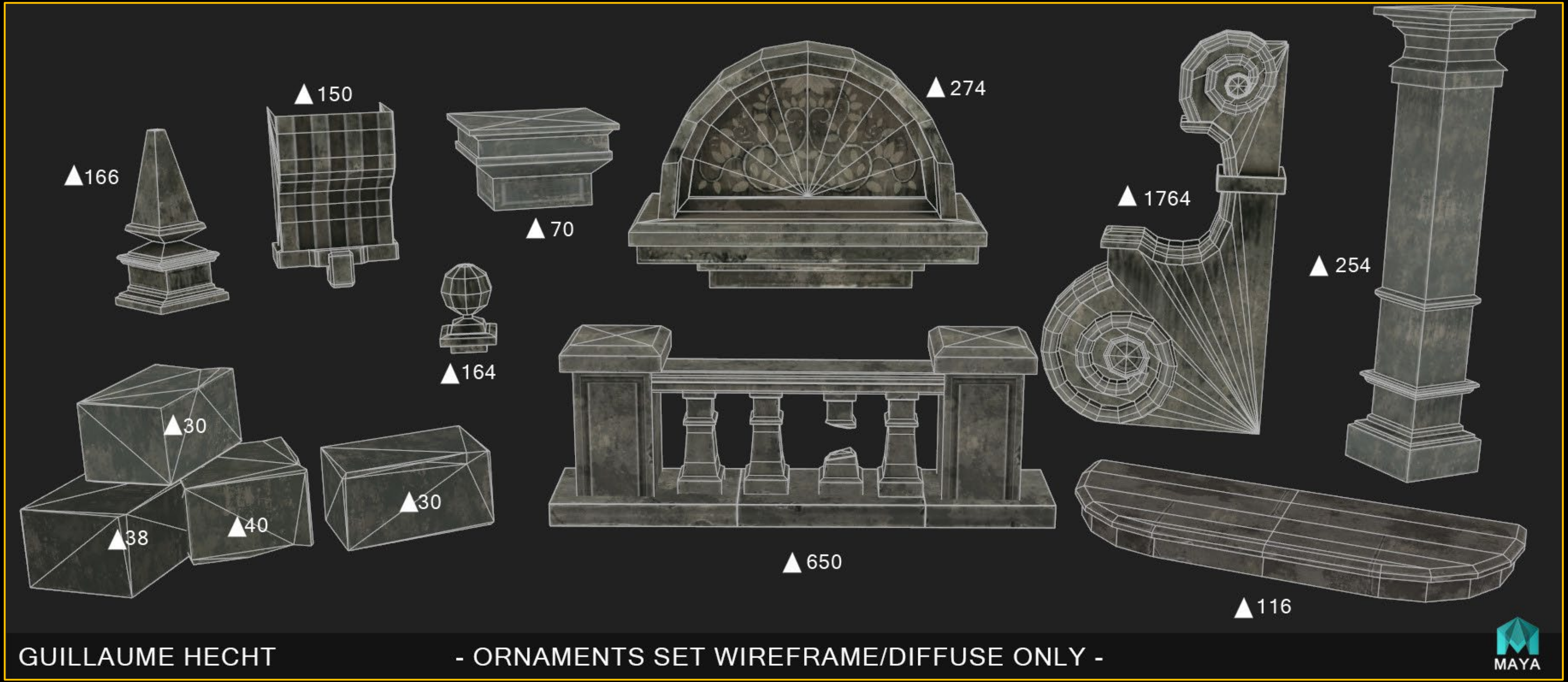
GRAPHICS PROGRAMMING I

TRIANGLE MESHES

Ray Tracing: Intersections – Ray-Triangle?

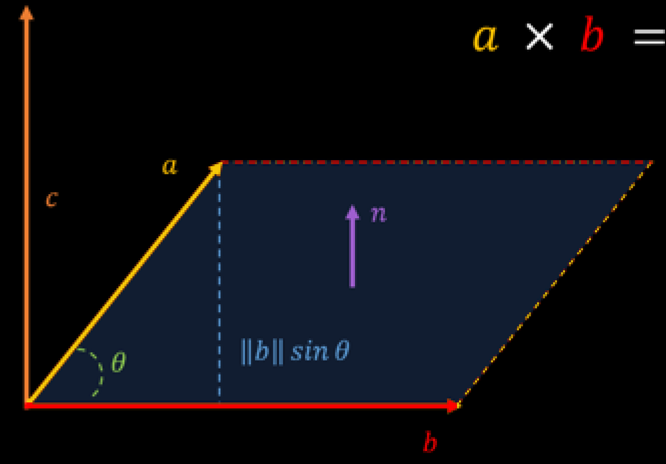
- Up until now we've used basic primitives (planes an spheres) using implicit equations. So how do we render (triangle) **meshes**?
- As you all know, in games, we represent our models as a collection of **triangles**. All we need to do is find a way to find intersections, from our ray, on a triangle.



Ray Tracing: Intersections – Ray-Triangle?

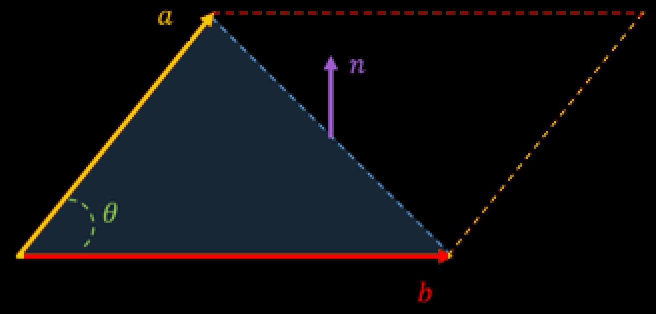
- Finding intersections between a ray and a triangle is not hard! ☺
- Remember, during the first class we've talked about the **cross product** and the fact that the cross product of two vectors gives us another vector which size is equal to the **area of the parallelogram** formed by those two vectors?
- A triangle is one half of a parallelogram. Let's discuss a simple ray-triangle intersection algorithm!

$$a \times b = \|a\|\|b\| \sin\theta \, n$$



$A = \|a\|\|b\| \sin\theta \longrightarrow A = \|a \times b\|$

$A = \|c\|$

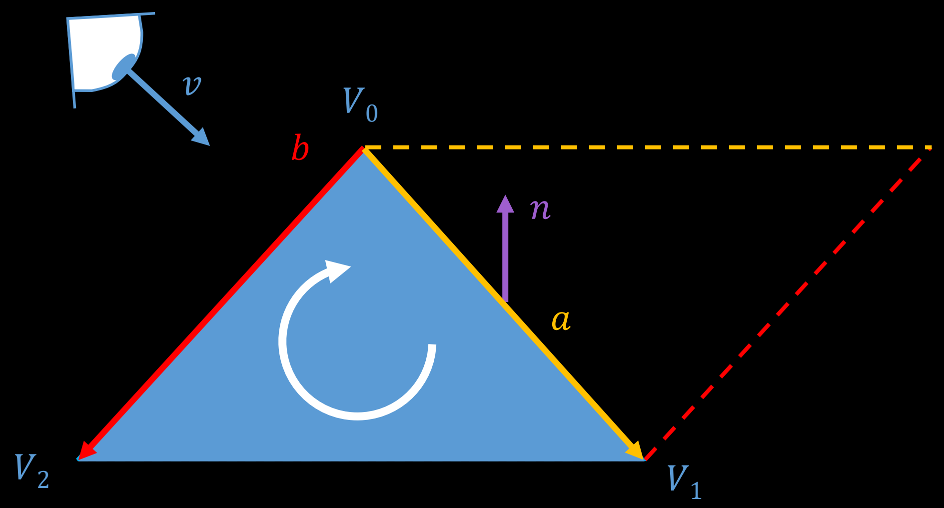


$A\,\text{triangle} = \frac{1}{2}\|a \times b\|$

Ray Tracing: Intersections – Ray-Triangle?

- A triangle consists out of 3 vertices. They should have a clock-wise order. This is due to our coordinate system (left-handed) and how we form vectors using the cross product.
- Using those 3 vertices we can construct 2 vectors (edges) and use those to determine the normal of our plane/triangle.
- We can start by checking if we intersect with our plane, almost like with the implicit plane equation.

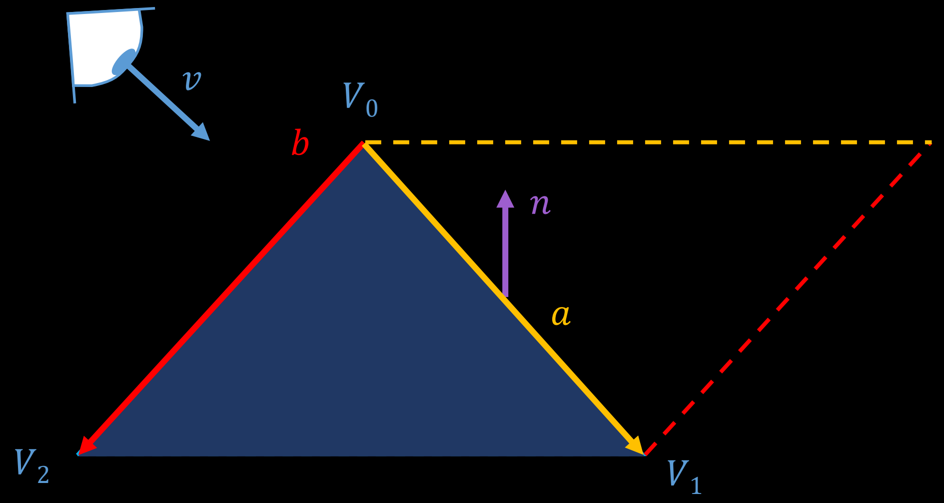
```
a = v1 - v0
b = v2 - v0
normal = Cross(a, b)

Dot(n,v) == 0
    return false
```



Ray Tracing: Intersections – Ray-Triangle?

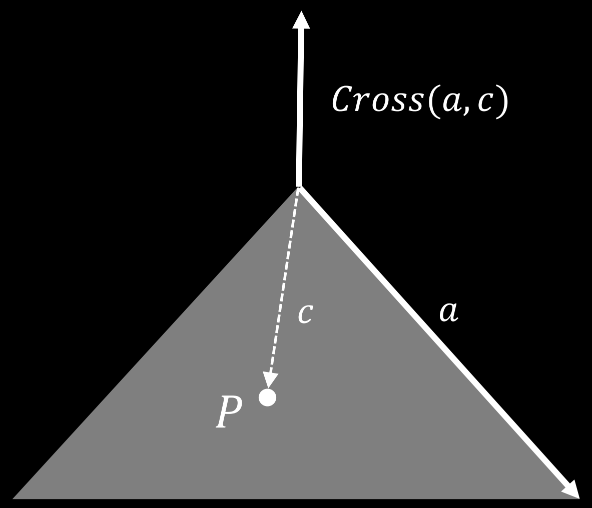
- Once we know we are hitting the plane, we can calculate our t value just as we did with ray-plane intersection. The center can be (0,0,0), but because vertices can have different depth values you should calculate the "center" ((v0+v1+v2)/3)
- You also check if it is within range of the ray.
- And you finally calculate the intersection point. Now we have the intersection point on the plane, not the triangle!

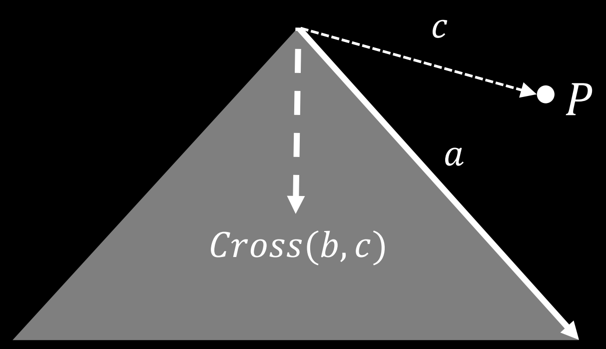
```
a = v1 - v0
b = v2 - v0
normal = Cross(a, b)

Dot(n,v) == 0
    return false

L = center - ray.origin
t = Dot(L, normal) /
Dot(v, normal)

t < tMin || t > tMax
   return false

p = ray.origin + t * v
```



Ray Tracing: Intersections – Ray-Triangle?

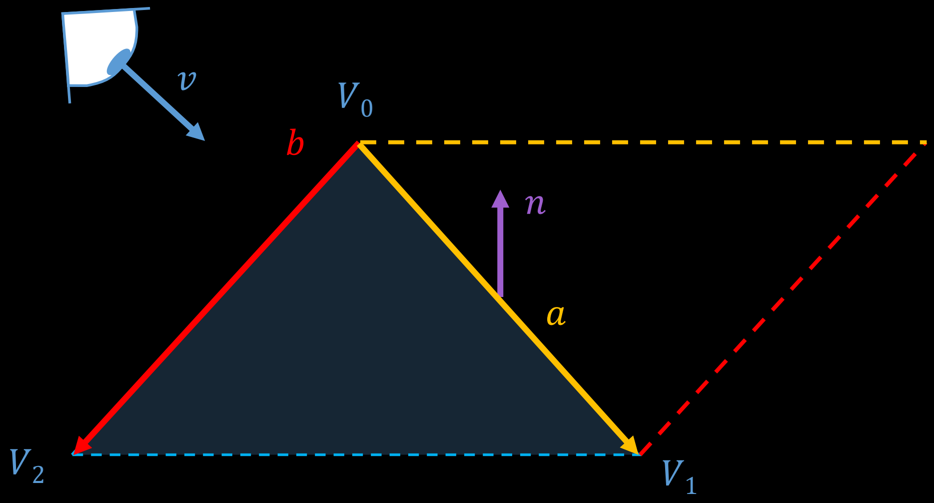
- To find out if a point is inside a triangle, we can use the power of the signed area of the cross product!
- When we create vector $\boldsymbol{c} = \boldsymbol{P} - \boldsymbol{V_0}$ and take the cross product between this vector and one of the edges, we get a vector perpendicular to the triangle, but depending on where the point P is, we get a different direction.
- We can then check if the normal and this cross-product points in the same direction using the dot product. If the dot product is bigger than 0, they point in the same direction!





Ray Tracing: Intersections – Ray-Triangle?

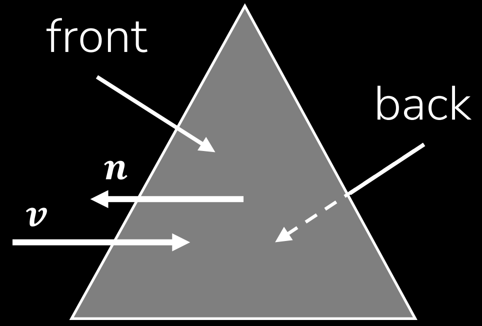
- We can now complete our test using the discussed technique for every edge. In other words, we check if the point $\boldsymbol{P}$ is on the "correct" side of the edge. If this is true for all edges, we know the point $\boldsymbol{P}$ is inside the triangle.
- This is one technique! It's the easiest to understand. There are other, more performant, techniques out there. Optimizing ray-triangle intersection is worth looking into. It's an active research topic!




```
a = v1 - v0
b = v2 - v0
normal = Cross(a, b)

Dot(n,v) == 0
    return false

L = center - ray.origin
t = Dot(L, normal) /
Dot(v, normal)

t < tMin || t > tMax
   return false

p = ray.origin + t * v

edgeA = v1 - v0
pointToSide = p - v0
Dot(normal, Cross(edgeA,
pointToSide) < 0
   return false
…

Fill in HitRecord
return true
```

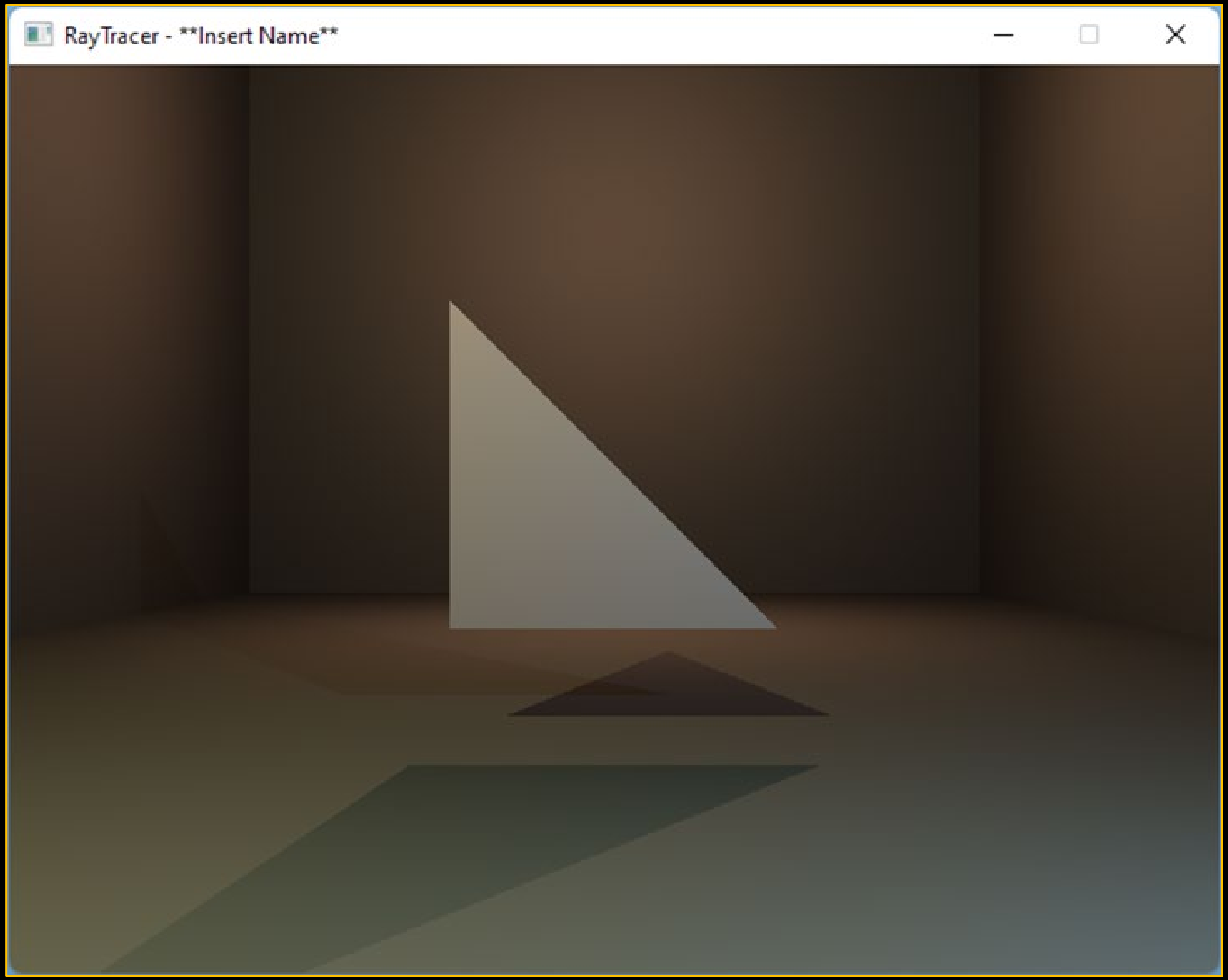
Ray Tracing: Intersections – Ray-Triangle?

- There is one more thing… Triangles, just as planes, have two sides. It has a front and a back side.
- Depending on where you hit the triangle, you want it to be visible or not. By default, it will be rendered from both sides. This is often referred to as double-sided primitives.
- In most renderers you have the option to select if you only want to see the front side, the back side or both sides. This is useful for optimization and certain effects.
- We also want this option. So, provide an option that is used in the intersection function, to determine if a triangle is visible or not. We will call this our cullmode (back-face, front-face or no culling). So, when we enable front-face culling for example, we don't want to render the front of a triangle! → culling == removing/reduction
- Be careful though! Depending on how you implemented shadows, you might want to use the opposite cullmode when calculating shadows! ☺



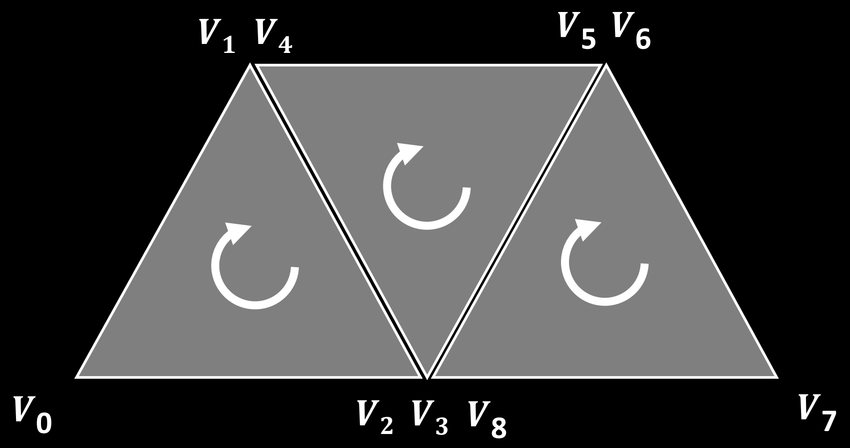

```
//Back-face
if(Dot(normal, v) > 0)

//Front-face
if(Dot(normal, v) < 0)
```

Ray Tracing: Single Triangle



Ray Tracing: Triangle Meshes

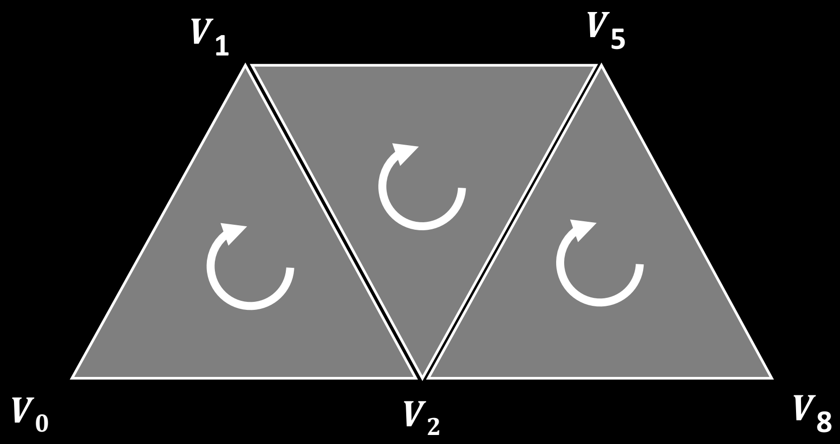
- Now that we can intersect with one triangle, let's have a look at how we are going to render a mesh containing multiple triangles.
- As you've could have guessed, rendering a mesh is just doing more ray-triangle intersection tests. But how do we store these triangles from our mesh?
- A straightforward implementation is to store all vertices…



V_0 = (-1, 0, 0)
V_1 = (-.5, 1, 0)
V_2 = (0, 0, 0)
V_3 = (0, 0, 0)
V_4 = (-.5, 1, 0)
V_5 = (.5, 1, 0)
V_6 = (.5, 1, 0)
V_7 = (1, 0, 0)
V_8 = (0, 0, 0)

Ray Tracing: Triangle Meshes

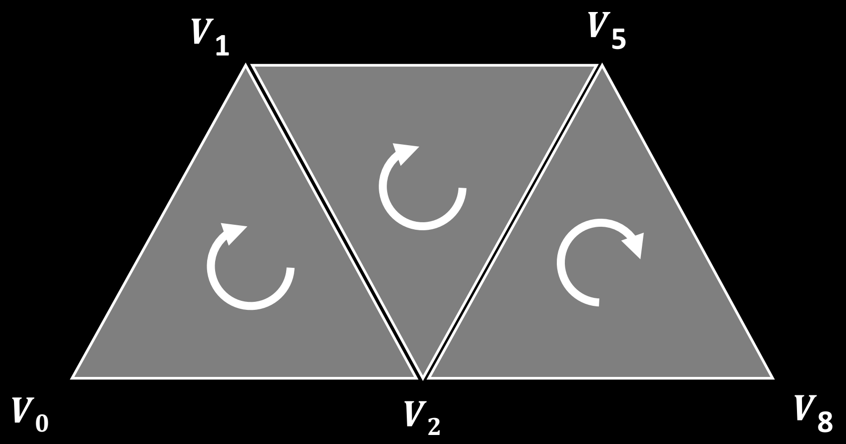
- Well, that is a waste of memory! In our case a vertex is just a position, but as we'll see later, it can hold more than just a position.
- As you've probably already noticed, there are a lot of duplicates. Let's fix this by deleting the duplicates.
  - In this case we just saved 48 bytes, if we store the individual values using floats!
- Now, how do we represent which vertex is used for which triangle?



V_0 = (-1, 0, 0)
V_1 = (-.5, 1, 0)
V_2 = (0, 0, 0)
V_5 = (.5, 1, 0)
V_8 = (1, 0, 0)

Ray Tracing: Triangle Meshes

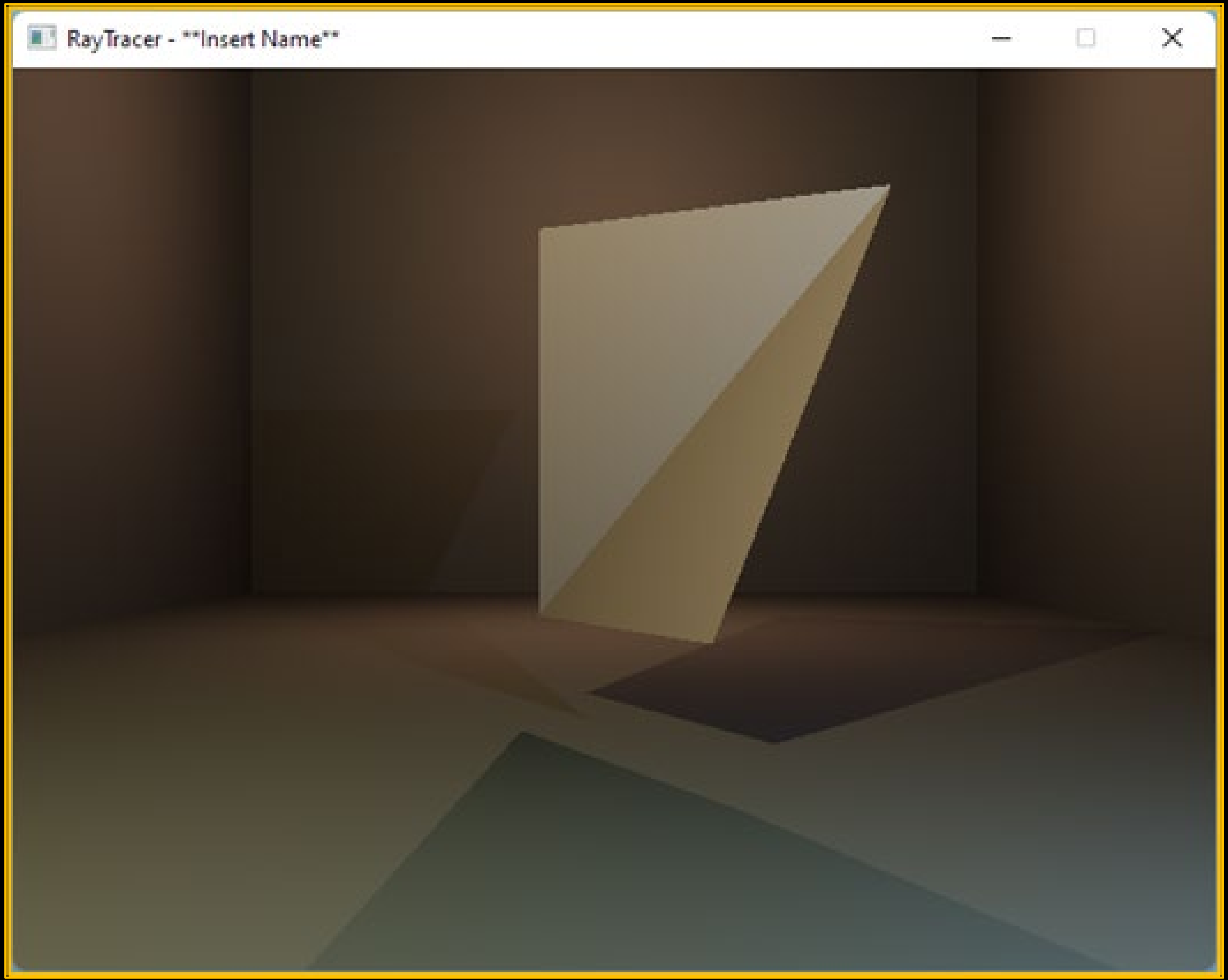
- If we give an ID to each vertex, we can actually represent each triangle by using an ID triplet.
- We can store the ID's as unsigned integers.
  - In this case we store 9 ID's, which has a total size of 36 bytes, so we still managed to save 12 bytes. As you can imagine, with meshes that have a lot of triangles, this can save a lot of memory.



V_0 = (-1, 0, 0) = 0
V_1 = (-.5, 1, 0) = 1
V_2 = (0, 0, 0) = 2
V_5 = (.5, 1, 0) = 3
V_8 = (1, 0, 0) = 4

T_0 = (0, 1, 2)
T_1 = (1, 3, 2)
T_2 = (2, 3, 4)

Ray Tracing: TriangleMesh



Ray Tracing: Triangle Meshes

- But what about “real” 3D triangle meshes?
- Most models are created using a DCC (Digital Content Creation) package like 3DS Max, Maya, Blenders, etc. You can export the models you’ve created in different formats: .obj, .fbx, .dae, etc.
  Most game engines support the import of different formats and parse it to a more engine friendly binary format (custom to each engine).
- Some formats are easier to parse than other! For example, parsing a binary .fbx file requires you to use an SDK (Software Development Kit) because it is a closed format. (Although, unofficial binary file format specifications can be found online)
- We want you to be able to import a simple .obj file.
  - Why: because it is easy and it’s just a text file, which can be read by a human.
- In order to be able to import this file you need to know the layout or architecture of the file. Let’s have a look…

Ray Tracing: OBJs

- The .obj file format can contain different pieces of information:
  - Vertex Positions
  - Faces (indices)
  - Vertex Normals → not face normal!
  - Vertex UV Coordinates
  - Smoothing Groups
  - …
- We only want you to support
  - Vertex Positions
  - Faces (indices)
- Feel free to have a look at the "full" documentation: https://en.wikipedia.org/wiki/Wavefront_.obj_file

```
# 3ds Max Wavefront OBJ Exporter v0.97b - (c)2007 guruware
# File Created: 22.10.2019 10:59:02

#
# object Box001
#

v  -0.0283 1.1837 -0.8430
v  0.4797 1.3464 -0.7806
v  0.1904 1.1592 -1.0725
v  0.4974 0.1568 0.7287
v  0.7526 0.2857 0.5918
v  0.7489 0.4989 0.7328
v  0.0284 3.2206 -1.0768
v  -0.2043 3.0557 -0.8937
v  -0.0189 3.1984 -0.6348
v  -0.4858 1.8106 0.2438
v  -0.5313 1.4169 0.5471
v  -0.0903 1.7853 0.3245
v  -0.2736 0.7921 -0.7241
v  -0.7374 1.3046 -0.8878
v  0.7952 1.1302 -0.8498
v  1.0129 0.8884 -0.7539
v  -1.4905 1.5743 0.4501
v  -1.5868 1.7742 0.5090
v  -1.5819 1.6510 -0.0459
o Box001
g Box001
f 1 2 3
f 4 5 6
f 7 8 9
f 10 11 12
f 1 13 14
f 15 16 3
f 17 18 19
f 20 21 22
f 23 24 25
f 26 27 20
f 28 29 30
f 27 31 32
f 33 34 29
f 35 36 37
f 38 39 40
f 41 42 43
f 44 24 23
```

Ray Tracing: OBJs

- The project already contains a fully implemented basic OBJ parser function!

- Utils::ParseOBJ

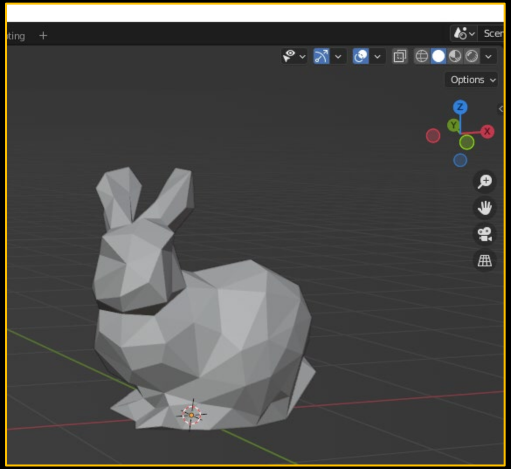
```cpp
namespace Utils
{
	//Just parses vertices and indices
#pragma warning(push)
#pragma warning(disable : 4505) //Warning unreferenced local function
	static bool ParseOBJ(const std::string& filename, std::vector<Vector3>& positions, std::vector<Vector3>& normals, std::vector<int>& indices)
	{
		std::ifstream file(filename);
		if (!file)
			return false;

		std::string sCommand;
		// start a while iteration ending when the end of file is reached (ios::eof)
		while (!file.eof())
		{
			//read the first word of the string, use the >> operator (istream::operator>>)
			file >> sCommand;
			//use conditional statements to process the different commands
			if (sCommand == "#")
			{
				// Ignore Comment
			}
			else if (sCommand == "v")
			{
				//Vertex
				float x, y, z;
				file >> x >> y >> z;
				positions.push_back({ x, y, z });
			}
			else if (sCommand == "f")
			{
				float i0, i1, i2;
```

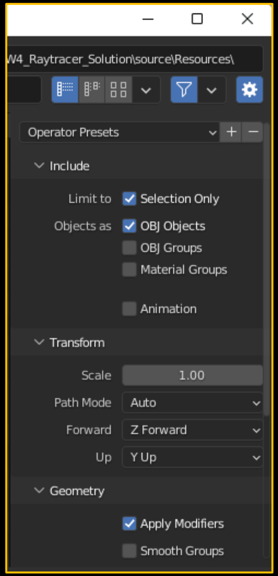
```
# 3ds Max Wavefront OBJ Exporter v0.97b - (c)2007 guruware
# File Created: 22.10.2019 10:59:02

#
# object Box001
#

v  -0.0283 1.1837 -0.8430
v  0.4797 1.3464 -0.7806
v  0.1904 1.1592 -1.0725
v  0.4974 0.1568 0.7287
v  0.7526 0.2857 0.5918
v  0.7489 0.4989 0.7328
v  0.0284 3.2206 -1.0768
v  -0.2043 3.0557 -0.8937
v  -0.0189 3.1984 -0.6348
v  -0.4858 1.8106 0.2438
v  -0.5313 1.4169 0.5471
v  -0.0903 1.7853 0.3245
v  -0.2736 0.7921 -0.7241
v  -0.7374 1.3046 -0.8878
v  0.7952 1.1302 -0.8498
v  1.0129 0.8884 -0.7539
v  -1.4905 1.5743 0.4501
v  -1.5868 1.7742 0.5090
v  -1.5819 1.6510 -0.0459
o Box001
g Box001
f 1 2 3
f 4 5 6
f 7 8 9
f 10 11 12
f 1 13 14
f 15 16 3
f 17 18 19
f 20 21 22
f 23 24 25
f 26 27 20
f 28 29 30
f 27 31 32
f 33 34 29
f 35 36 37
f 38 39 40
f 41 42 43
f 44 24 23
```

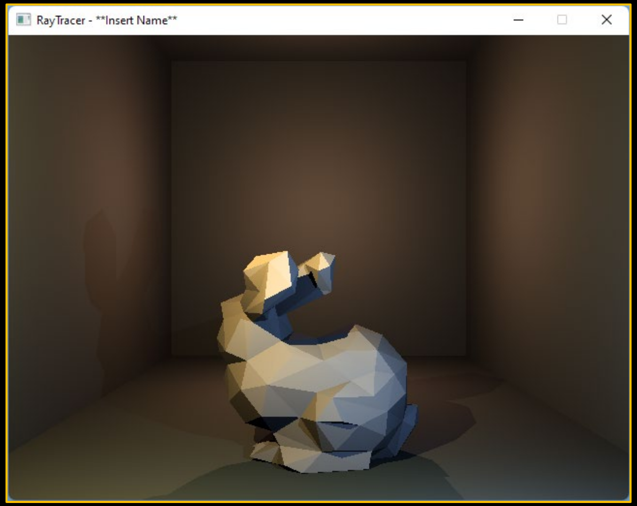
Ray Tracing: OBJs

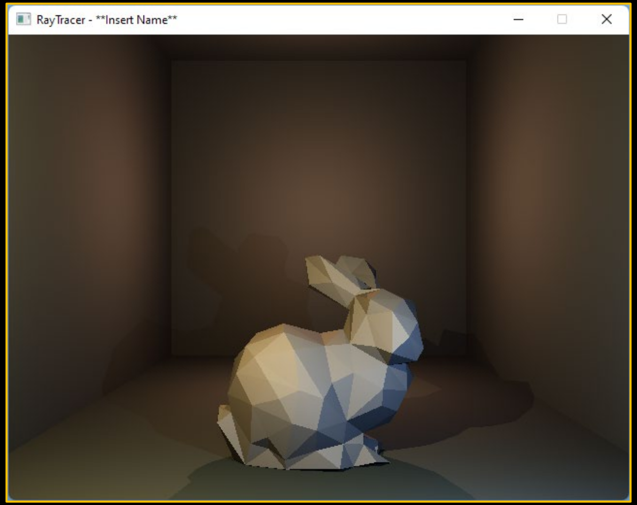
- The in-engine orientation/representation of a model can be different than visualized in your CCD (like Blender) package.
- Be aware of this!

- This depends on
  - Triangle Winding Order (CW vs CCW)
  - Axis System (Left- vs Right-handed)
  - Up axis (Z- vs Y- UP)
  - Exporting Options…



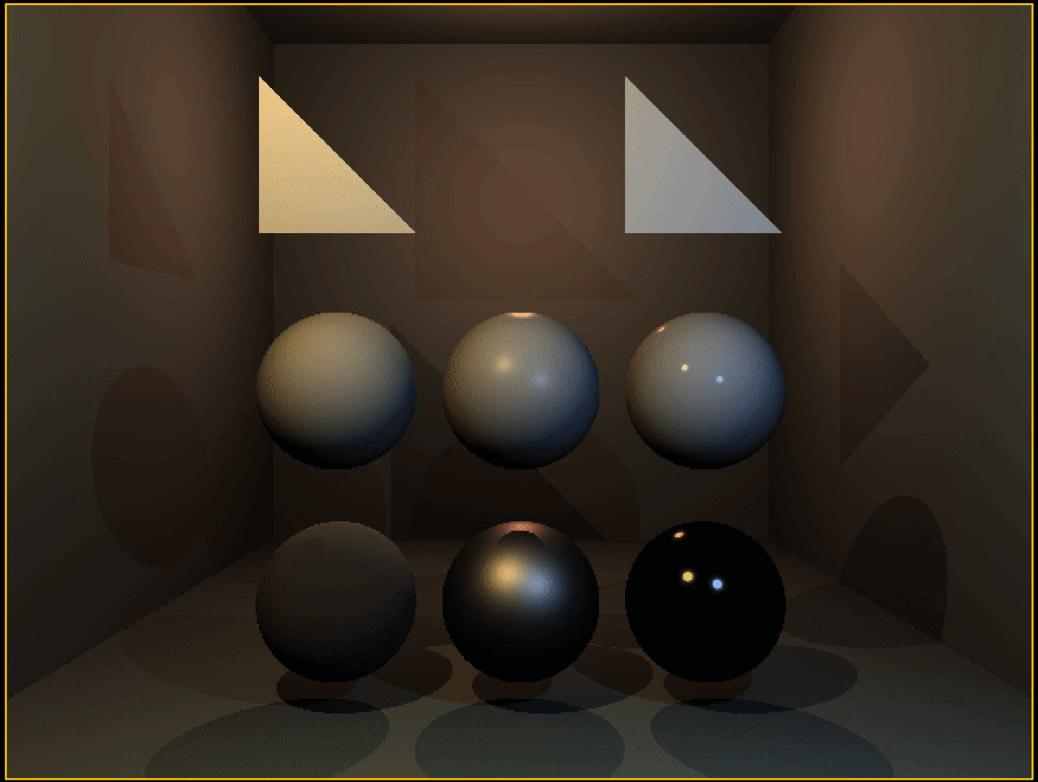






Ray Tracing: Triangle Meshes

- With all this knowledge you should be able to create the two final scenes
  - Reference Scene
  - Bunny Scene



GOD OF WAR

Vista by Timo Pihlajamaki & Kyle Bromley

Aaron Contreras/ Sr. Environment Artist

Ray Tracing: What’s next?

- With this lab you have all the bits and pieces for a very basic raytracing application. We have only scratched the surface of ray tracing. You can keep on improving your ray tracer. For people who are interested, these are some things you can investigate:
    - Multisampling (anti aliasing, etc.)
    - Reflections
    - Area Lights
    - Soft Shadows
    - Transparency
    - Global (Indirect) Illumination
    - Texture Mapping
    - Caustics
    - Optimizations:
        - Acceleration Structures
        - Multithreading (concept next week)
        - Ray binning
        - Better Data Oriented Design
    - …


GEFORCE
RTX™
MINECRAFT
NVIDIA®

Raytracing | Implementation Todos

1. Implement Triangle
    1. Use Scene_W4_TestScene for testing
    2. Temporarily use a vector<Triangle>
    3. Implement GeometryUtils::HitTest_Triangle

2. Implement TriangleMesh
    1. Use Scene_W4_TestScene for testing
    2. Implement TriangleMesh::CalculateNormals
    3. Implement GeometryUtils::HitTest_TriangeMesh
    4. Implement TriangleMesh::UpdateTransforms

3. Implement Scene_W4_ReferenceScene
4. Implement Scene_W4_BunnyScene

Raytracing | Implement Triangles (1)

- The Triangle primitive is already defined and implemented (DataTypes.h)

- Create a new Scene (Scene_W4_TestScene)
  - Temporarily add a vector<Triangle> to the (base) Scene class
  - Add the logic to iterate and test Triangles in Scene::GetClosestHit & Scene::DoesHit – use GeometryUtils::HitTest_Triangle (not implemented yet)

```cpp
void Scene_W4_TestScene::Initialize()
{
	m_Camera.origin = { 0.f, 1.f, -5.f };
	m_Camera.fovAngle = 45.f;

	//Materials
	const auto matLambert_GrayBlue = AddMaterial(new Material_Lambert({ .49f, 0.57f, 0.57f }, 1.f));
	const auto matLambert_White = AddMaterial(new Material_Lambert(colors::White, 1.f));

	//Planes
	AddPlane(Vector3{ 0.f, 0.f, 10.f }, Vector3{ 0.f, 0.f, -1.f }, matLambert_GrayBlue); //BACK
	AddPlane(Vector3{ 0.f, 0.f, 0.f }, Vector3{ 0.f, 1.f, 0.f }, matLambert_GrayBlue); //BOTTOM
	AddPlane(Vector3{ 0.f, 10.f, 0.f }, Vector3{ 0.f, -1.f, 0.f }, matLambert_GrayBlue); //TOP
	AddPlane(Vector3{ 5.f, 0.f, 0.f }, Vector3{ -1.f, 0.f, 0.f }, matLambert_GrayBlue); //RIGHT
	AddPlane(Vector3{ -5.f, 0.f, 0.f }, Vector3{ 1.f, 0.f, 0.f }, matLambert_GrayBlue); //LEFT

	//Triangle (Temp)
	auto triangle = Triangle{ {-.75f,.5f,.0f},{-.75f,2.f, .0f}, {.75f,.5f,0.f} };
	triangle.cullMode = TriangleCullMode::NoCulling;
	triangle.materialIndex = matLambert_White;

	m_Triangles.emplace_back(triangle);

	//Light
	AddPointLight(Vector3{ 0.f, 5.f, 5.f }, 50.f, ColorRGB{ 1.f, .61f, .45f }); //Backlight
	AddPointLight(Vector3{ -2.5f, 5.f, -5.f }, 70.f, ColorRGB{ 1.f, .8f, .45f }); //Front Light Left
	AddPointLight(Vector3{ 2.5f, 2.5f, -5.f }, 50.f, ColorRGB{ .34f, .47f, .68f });
}
```

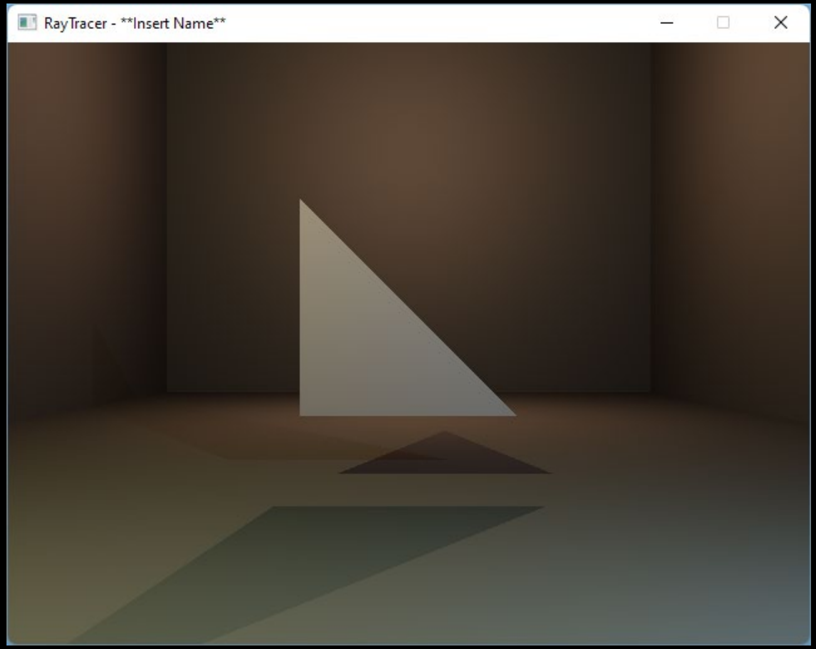
Scene class (Scene.h)

```cpp
std::vector<Material*> m_Materials{};

//Temp (Individual Triangle Testing)
std::vector<Triangle> m_Triangles{};

Camera m_Camera{};
```

Raytracing | Implement Triangles (2)

- Implement the GeometryUtils::HitTest_Triangle(…) function
  - Normal VS Ray-Direction Check (Perpendicular?)
  - Cull Mode Check
    - Based on the Cull-Mode the Triangle is visible or invisible (culled), keep in mind the cull-mode must be inverted for the shadow-rays. (Hint: We can assume that if 'ignoreHitRecord' is TRUE that we are performing a shadow hittest…)
  - Ray-Plane test (plane defined by Triangle) + T range check
  - Check if hitpoint is inside the Triangle
  - Fill-in HitRecord (if required)

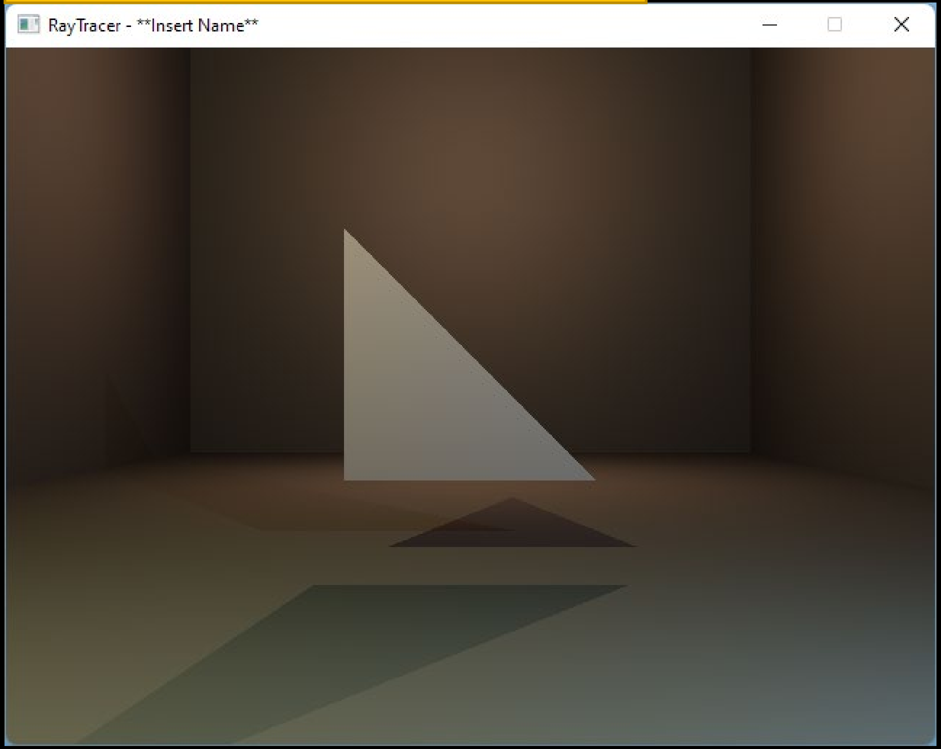
  - Use the parameters in the next slides to verify you implementation
    - No Culling
    - Front-Face Culling
    - Back-Face Culling



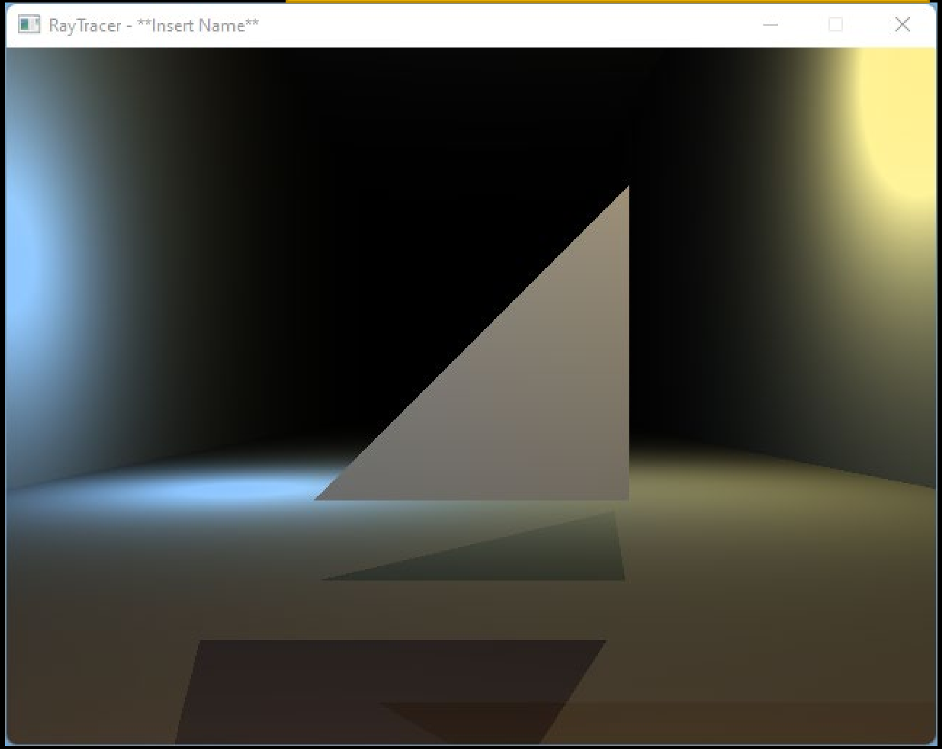
Raytracing | Verify Triangles (3)

NO
Culling

Camera Origin {.0f, 1.f, -5.f}
Camera TotalYaw {0}



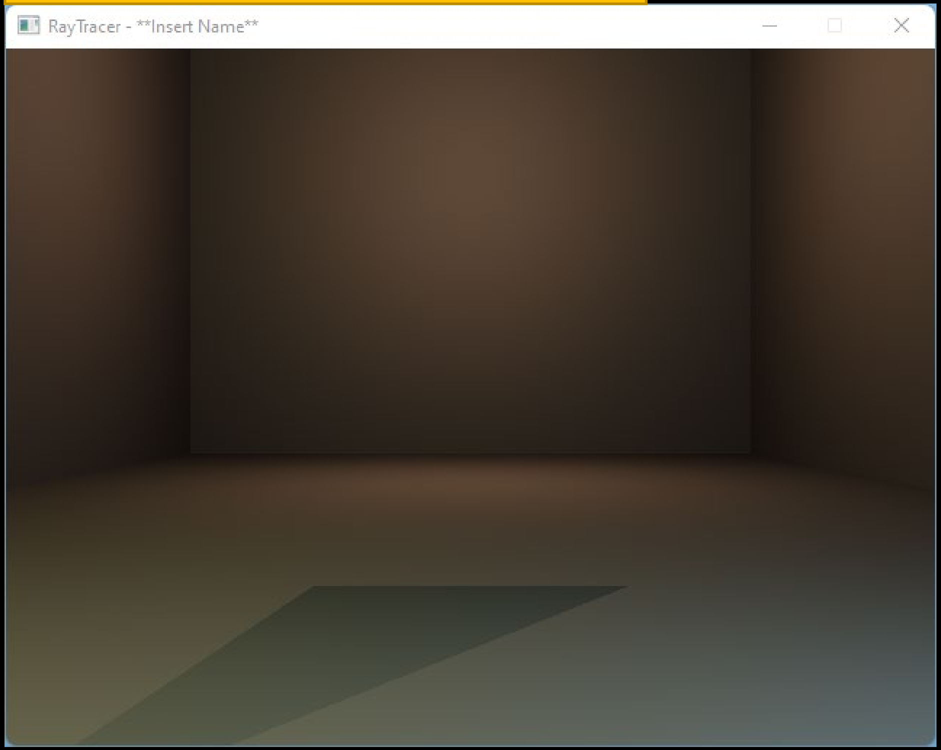
Camera Origin {.0f, 1.f, 4.f}
Camera TotalYaw {PI}



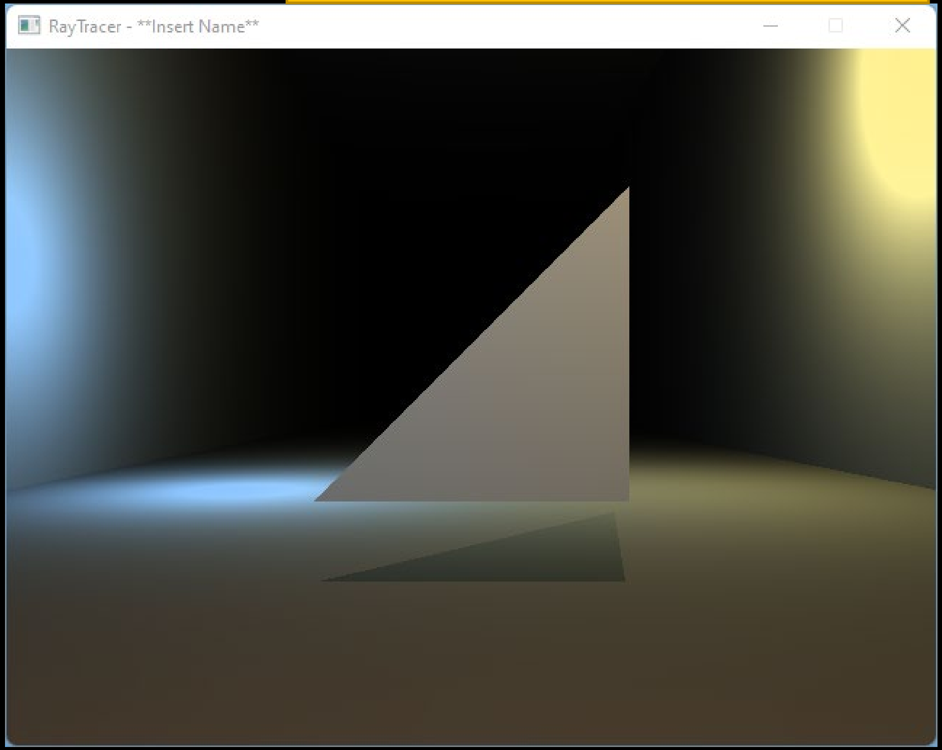
Raytracing | Verify Triangles (4)

FRONT FACE
Culling

Camera Origin {.0f, 1.f, -5.f}
Camera TotalYaw {0}



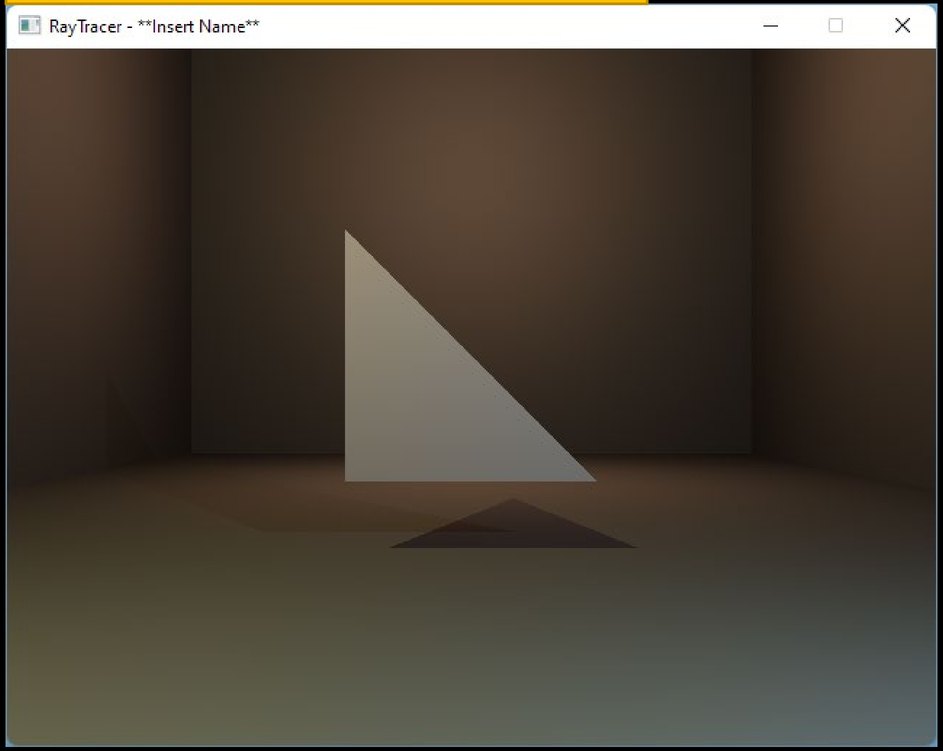
Camera Origin {.0f, 1.f, 4.f}
Camera TotalYaw {PI}



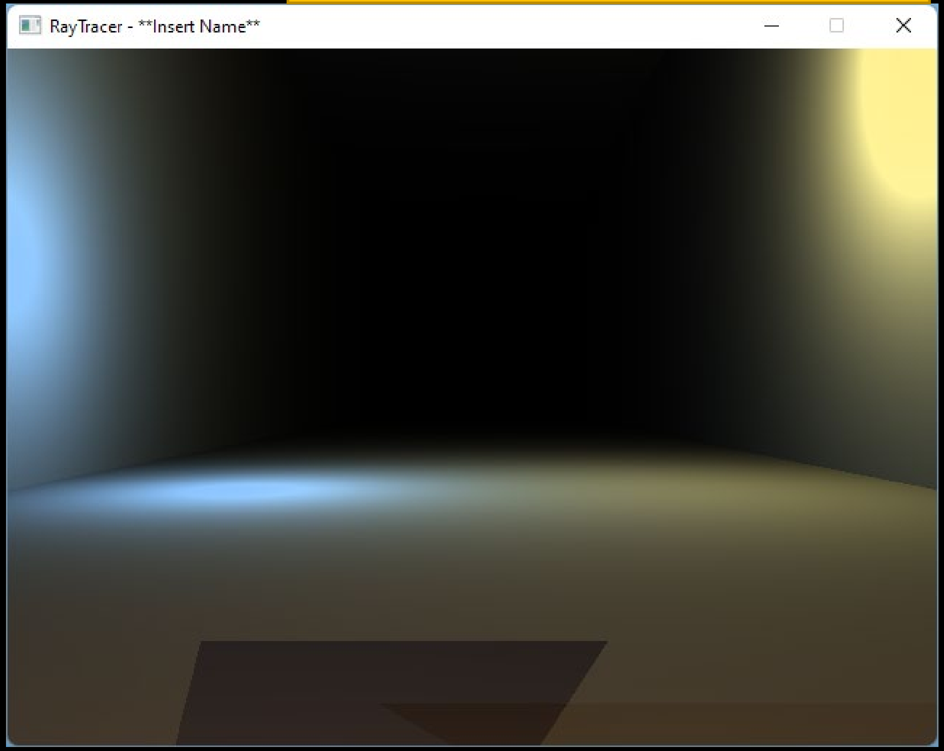
Raytracing | Verify Triangles (5)

BACK FACE Culling

Camera Origin {.0f, 1.f, -5.f}
Camera TotalYaw {0}



Camera Origin {.0f, 1.f, 4.f}
Camera TotalYaw {PI}



Raytracing | Implement TriangleMesh (6)

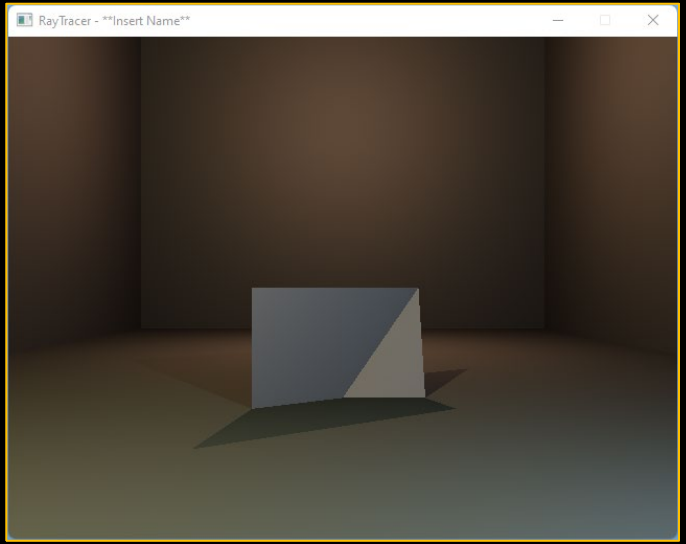
- The TriangleMesh primitive is partially defined (DataTypes.h)
  - TriangleMesh::CalculateNormals (TODO)
    - Should calculate the normal for each triangle defined by the Positions & Indices buffers, store the results in 'normals'
  - TriangleMesh::UpdateTransforms (TODO)
    - Should transform the positions & normals (translation, rotation, scale matrices) and store the result in 'transformedPositions' & 'transformedNormals' respectively

- Alter Scene_W4_TestScene (using TriangleMesh instead of Triangle)
- Remove Temporary Triangle Vector in Scene Class (also the Triangle hittest logic in Scene::GetClosestHit & Scene::DoesHit – should be replaced with TriangleMesh hittest logic)

```
////Triangle (Temp)
//auto triangle = Triangle{ {-.75f,.5f,.0f},{-.75f,2.f, .0f}, {.75f,.5f,0.f} };
//triangle.cullMode = TriangleCullMode::NoCulling;
//triangle.materialIndex = matLambert_White;

//m_Triangles.emplace_back(triangle);

//Triangle Mesh
const auto triangleMesh = AddTriangleMesh(TriangleCullMode::NoCulling, matLambert_White);
triangleMesh->positions = { {-.75f,-1.f,.0f},{-.75f,1.f, .0f}, {.75f,1.f,1.f}, {.75f,-1.f,0.f} };
triangleMesh->indices = {
    0,1,2, //Triangle 1
    0,2,3  //Triangle 2
};

triangleMesh->CalculateNormals();
triangleMesh->UpdateTransforms();
```

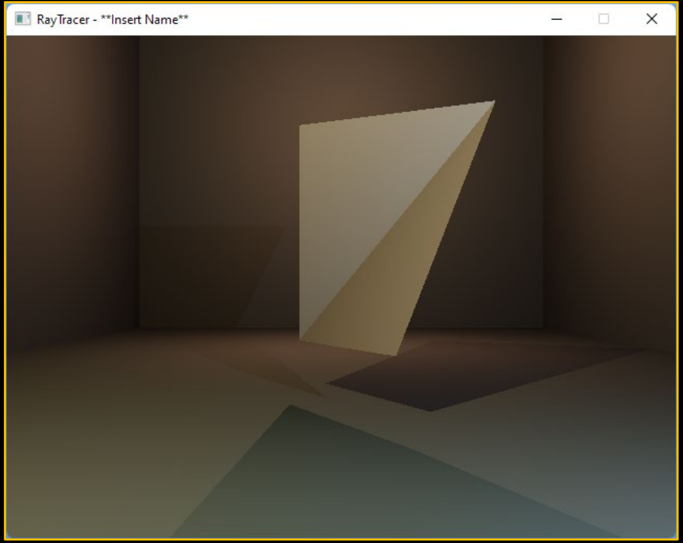
Raytracing | Implement TriangleMesh (6)

- Implement TriangleMesh::CalculateNormals (One normal per triangle)
- Implement TriangleMesh::UpdateTransforms >> Only copy the ‘positions’ & ‘normals’ into ‘transformedPositions’ & ‘transformedNormals’ respectively (for now)
- Implement the HitTest_TriangleMesh function (+ make sure to alter the Scene::GetClosestHit & DoesHit functions)
  - Each set of 3 indices represents a Triangle – use HitTest_Triangle to find the triangle of the TriangleMesh with the (!) closest hit
  - Use the ‘transformedPositions’ & ‘transformedNormals’ to define each individual triangle!



Raytracing | Implement TriangleMesh (7)

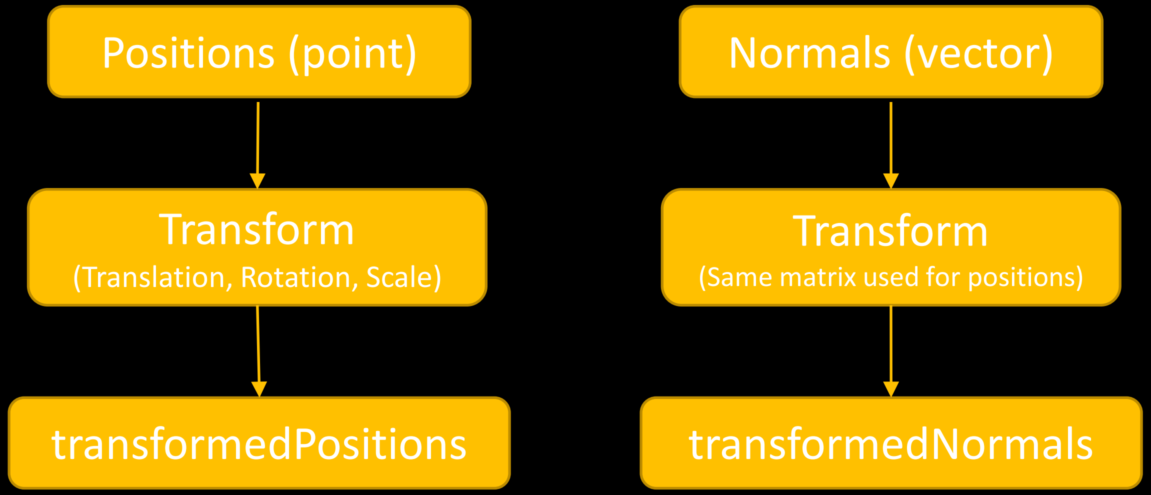
- Next step is to transform the TriangleMesh positions & normals based on the transformation matrices stored in the TriangleMesh primitive. (TriangleMesh::UpdateTransforms)
- Updating the positions & normals must happen each time one of the transformation components (translation, rotation or scale) are altered! (Order of operations is important here!!)
- Make sure you're code is 'optimized', make use of vector::reserve & vector::emplace_back

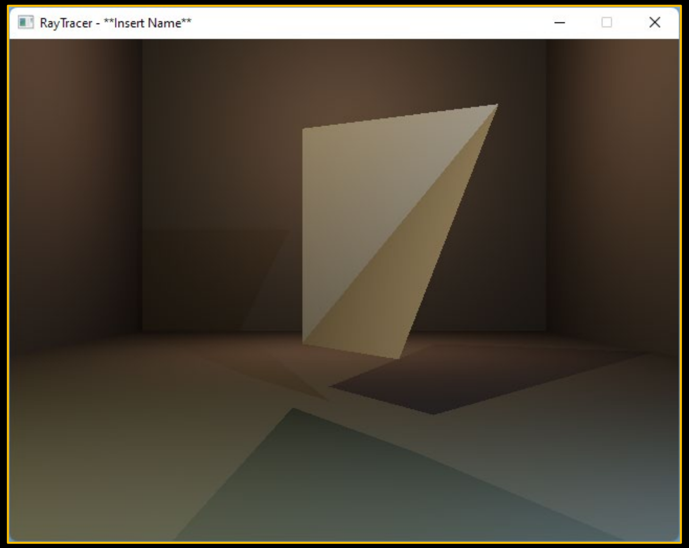



```cpp
//Triangle Mesh
const auto triangleMesh = AddTriangleMesh(TriangleCullMode::NoCulling, matLambert_White);
triangleMesh->positions = {
    {-.75f, -1.f, .0f},  //V0
    {-.75f, 1.f, .0f},   //V2
    {.75f, 1.f, 1.f},    //V3
    {.75f, -1.f, 0.f} }; //V4

triangleMesh->indices = {
    0,1,2, //Triangle 1
    0,2,3  //Triangle 2
};

triangleMesh->CalculateNormals();

triangleMesh->Translate({ 0.f, 1.5f, 0.f });
triangleMesh->RotateY(45);

triangleMesh->UpdateTransforms();
```

Raytracing | Implement TriangleMesh (8)

- Next step is to transform the TriangleMesh positions & normals based on the transformation matrices stored in the TriangleMesh primitive. (TriangleMesh::UpdateTransforms)
- Updating the positions & normals must happen each time one of the transformation components (translation, rotation or scale) are altered! (Order of operations is important here!!)
- Make sure you're code is 'optimized', make use of vector::reserve & vector::emplace_back





Raytracing | Implement TriangleMesh (9)

- Using the Update function from the Scene, we can now update the rotation of our TriangleMesh frame by frame
  - Override the base Scene Update Function
  - Store the TriangleMesh as a datamember
  - Update the rotation frame by frame
    - Do not forget to call the base Scene::Update function, otherwise your camera won't be updated anymore

Override Update + Store TriangeMesh

```
//+++++++++++++++++++++++++++++++++++++++++++
//WEEK 4 Test Scene
class Scene_W4_TestScene final : public Scene
{
public:
    Scene_W4_TestScene() = default;
    ~Scene_W4_TestScene() override = default;

    Scene_W4_TestScene(const Scene_W4_TestScene&) = delete;
    Scene_W4_TestScene(Scene_W4_TestScene&&) noexcept = delete;
    Scene_W4_TestScene& operator=(const Scene_W4_TestScene&) = delete;
    Scene_W4_TestScene& operator=(Scene_W4_TestScene&&) noexcept = delete;

    void Initialize() override;
    void Update(Timer* pTimer) override;

private:
    TriangleMesh* pMesh{ nullptr };
};
```

Keep track of the TriangleMesh (pMesh)

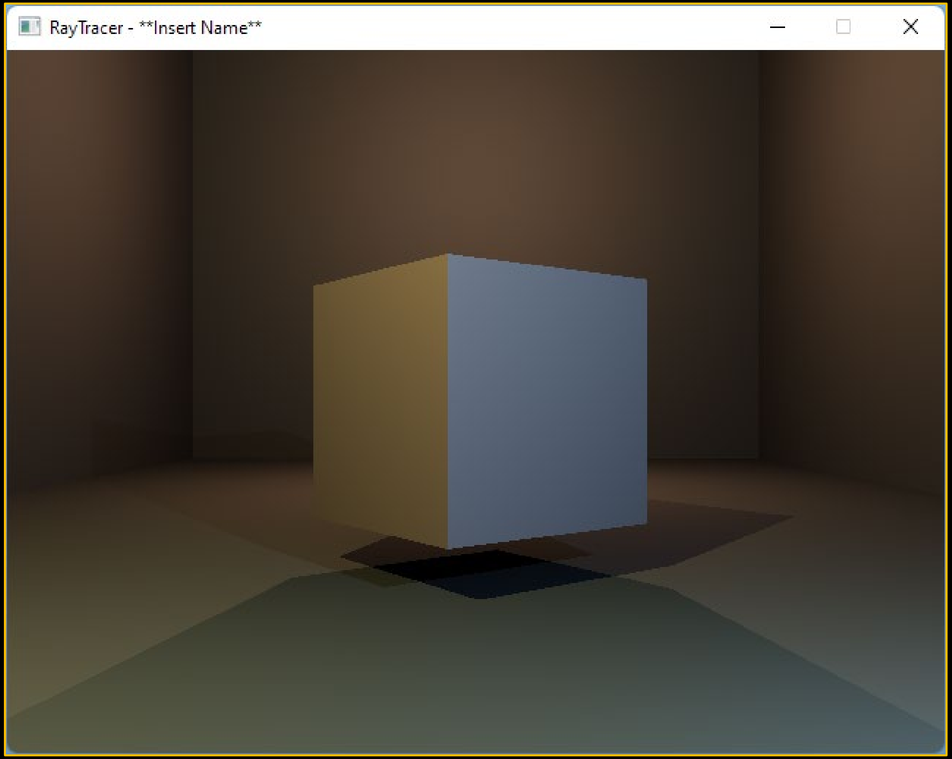
```
//Triangle Mesh
pMesh = AddTriangleMesh(TriangleCullMode::NoCulling, matLambert_White);
pMesh->positions = {
    {-.75f,-1.f,.0f},  //V0
    {-.75f,1.f, .0f},  //V2
    {.75f,1.f,1.f},    //V3
    {.75f,-1.f,0.f} }; //V4

pMesh->indices = {
    0,1,2, //Triangle 1
    0,2,3  //Triangle 2
};

pMesh->CalculateNormals();

pMesh->Translate({ 0.f,1.5f,0.f });
pMesh->UpdateTransforms();
```

Update the rotation + call Base Update (!)

```
void Scene_W4_TestScene::Update(Timer* pTimer)
{
    Scene::Update(pTimer);

    pMesh->RotateY(PI_DIV_2 * pTimer->GetTotal());
    pMesh->UpdateTransforms();
}
```

Raytracing | Implement TriangleMesh (10)

- Now we can parse the vertex information for an OBJ file to render more complex (store as OBJ) objects to the screen (Utils::ParseOBJ)
  - The ParseOBJ function parses a given .OBJ file and fills in a vector with position, normal and index information, which in turn can be used to render a TriangleMesh.



Use these simple objects (cube, box, object) to further debug any remaining rendering issues

```
pMesh = AddTriangleMesh(TriangleCullMode::BackFaceCulling, matLambert_White);
Utils::ParseOBJ("Resources/simple_cube.obj",
//Utils::ParseOBJ("Resources/simple_object.obj",
    [&] pMesh->positions,
    [&] pMesh->normals,
    [&] pMesh->indices);

pMesh->Scale({ _x: .7f, _y: .7f, _z: .7f });
pMesh->Translate(translation: { _x: .0f, _y: 1.f, _z: 0.f });

//No need to Calculate the normals, these are calculated inside the ParseOBJ function
pMesh->UpdateTransforms();
```

Raytracing | Reference Scene (11)

- Now we have all the bits and pieces to create the final Reference Scene

```cpp
void Scene_W4_ReferenceScene::Initialize()
{
    sceneName = "Reference Scene";
    m_Camera.origin = { _x:0, _y:3, _z:-9 };
    m_Camera.fovAngle = 45.f;

    const auto matCT_GrayRoughMetal :unsigned char = AddMaterial(new Material_CookTorrence( albedo: { .r:.972f, .g:.960f, .b:.915f }, metalness:1.f, roughness:1.f));
    const auto matCT_GrayMediumMetal :unsigned char = AddMaterial(new Material_CookTorrence( albedo: { .r:.972f, .g:.960f, .b:.915f }, metalness:1.f, roughness:.6f));
    const auto matCT_GraySmoothMetal :unsigned char = AddMaterial(new Material_CookTorrence( albedo: { .r:.972f, .g:.960f, .b:.915f }, metalness:1.f, roughness:.1f));
    const auto matCT_GrayRoughPlastic :unsigned char = AddMaterial(new Material_CookTorrence( albedo: { .r:.75f, .g:.75f, .b:.75f }, metalness:.0f, roughness:1.f));
    const auto matCT_GrayMediumPlastic :unsigned char = AddMaterial(new Material_CookTorrence( albedo: { .r:.75f, .g:.75f, .b:.75f }, metalness:.0f, roughness:.6f));
    const auto matCT_GraySmoothPlastic :unsigned char = AddMaterial(new Material_CookTorrence( albedo: { .r:.75f, .g:.75f, .b:.75f }, metalness:.0f, roughness:.1f));

    const auto matLambert_GrayBlue :unsigned char = AddMaterial(new Material_Lambert( diffuseColor: { .r:.49f, .g:0.57f, .b:0.57f }, diffuseReflectance:1.f));
    const auto matLambert_White :unsigned char = AddMaterial(new Material_Lambert(colors::White, diffuseReflectance:1.f));

    AddPlane( origin:Vector3{ _x:0.f, _y:0.f, _z:10.f }, normal:Vector3{ _x:0.f, _y:0.f, _z:-1.f }, matLambert_GrayBlue); //BACK
    AddPlane( origin:Vector3{ _x:0.f, _y:0.f, _z:0.f }, normal:Vector3{ _x:0.f, _y:1.f, _z:0.f }, matLambert_GrayBlue); //BOTTOM
    AddPlane( origin:Vector3{ _x:0.f, _y:10.f, _z:0.f }, normal:Vector3{ _x:0.f, _y:-1.f, _z:0.f }, matLambert_GrayBlue); //TOP
    AddPlane( origin:Vector3{ _x:5.f, _y:0.f, _z:0.f }, normal:Vector3{ _x:-1.f, _y:0.f, _z:0.f }, matLambert_GrayBlue); //RIGHT
    AddPlane( origin:Vector3{ _x:-5.f, _y:0.f, _z:0.f }, normal:Vector3{ _x:1.f, _y:0.f, _z:0.f }, matLambert_GrayBlue); //LEFT

    AddSphere(Vector3{ _x:-1.75f, _y:1.f, _z:0.f }, radius:.75f, matCT_GrayRoughMetal);
    AddSphere(Vector3{ _x:0.f, _y:1.f, _z:0.f }, radius:.75f, matCT_GrayMediumMetal);
    AddSphere(Vector3{ _x:1.75f, _y:1.f, _z:0.f }, radius:.75f, matCT_GraySmoothMetal);
    AddSphere(Vector3{ _x:-1.75f, _y:3.f, _z:0.f }, radius:.75f, matCT_GrayRoughPlastic);
    AddSphere(Vector3{ _x:0.f, _y:3.f, _z:0.f }, radius:.75f, matCT_GrayMediumPlastic);
    AddSphere(Vector3{ _x:1.75f, _y:3.f, _z:0.f }, radius:.75f, matCT_GraySmoothPlastic);
```

…

Raytracing | Reference Scene

- Now we have all the bits and pieces to create the final Reference Scene

```cpp
    //CW Winding Order!
    const Triangle baseTriangle = { Vector3(_x:-.75f, _y:1.5f, _z:0.f), Vector3(_x:.75f, _y:0.f, _z:0.f), Vector3(_x:-.75f, _y:0.f, _z:0.f) };

    m_Meshes[0] = AddTriangleMesh(TriangleCullMode::BackFaceCulling, matLambert_White);
    m_Meshes[0]->AppendTriangle(baseTriangle, ignoreTransformUpdate:true);
    m_Meshes[0]->Translate(translation:{ _x:-1.75f, _y:4.5f, _z:0.f });
    m_Meshes[0]->UpdateTransforms();

    m_Meshes[1] = AddTriangleMesh(TriangleCullMode::FrontFaceCulling, matLambert_White);
    m_Meshes[1]->AppendTriangle(baseTriangle, ignoreTransformUpdate:true);
    m_Meshes[1]->Translate(translation:{ _x:0.f, _y:4.5f, _z:0.f });
    m_Meshes[1]->UpdateTransforms();        [No Title]

    m_Meshes[2] = AddTriangleMesh(TriangleCullMode::NoCulling, matLambert_White);
    m_Meshes[2]->AppendTriangle(baseTriangle, ignoreTransformUpdate:true);
    m_Meshes[2]->Translate(translation:{ _x:1.75f, _y:4.5f, _z:0.f });
    m_Meshes[2]->UpdateTransforms();

    AddPointLight(Vector3{ _x:0.f, _y:5.f, _z:5.f }, intensity:50.f, ColorRGB{ _r:1.f, _g:.61f, _b:.45f }); //Backlight
    AddPointLight(Vector3{ _x:-2.5f, _y:5.f, _z:-5.f }, intensity:70.f, ColorRGB{ _r:1.f, _g:.8f, _b:.45f }); //Front Light Left
    AddPointLight(Vector3{ _x:2.5f, _y:2.5f, _z:-5.f }, intensity:50.f, ColorRGB{ _r:.34f, _g:.47f, _b:.68f });
}
```

Raytracing | Reference Scene

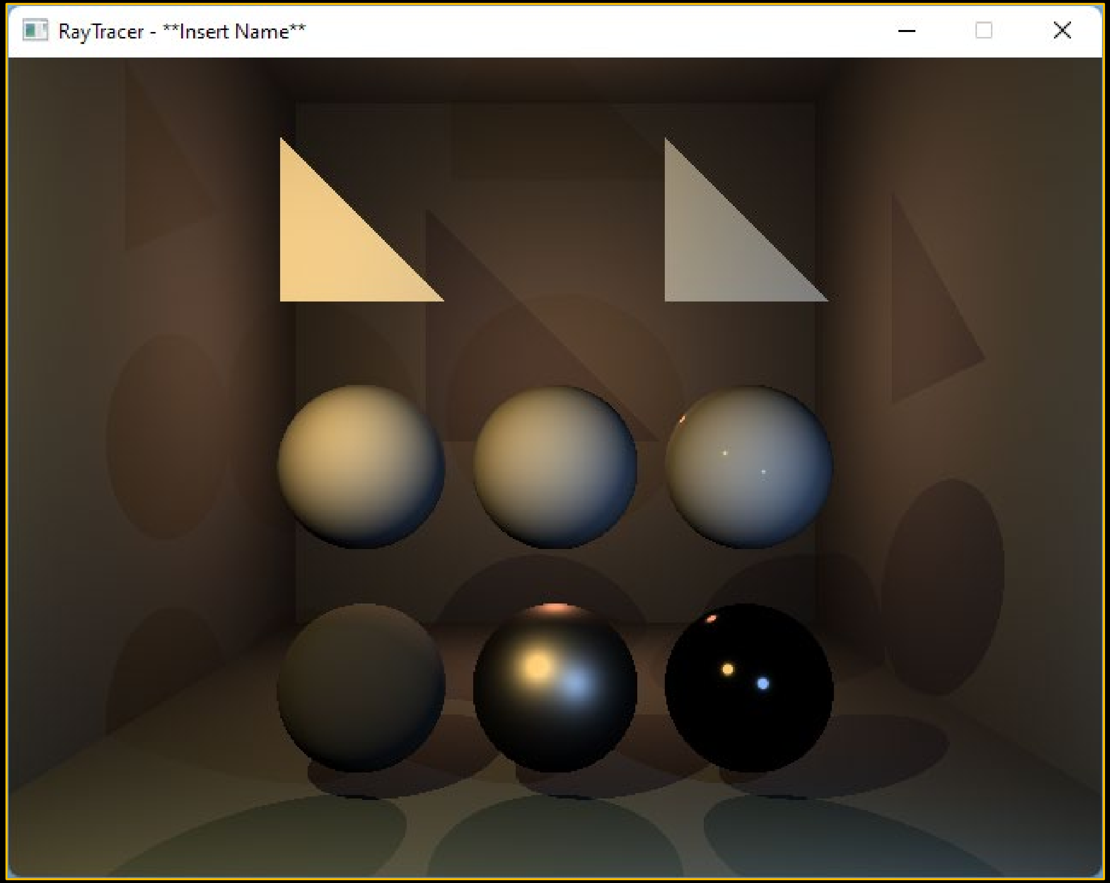
- Now we have all the bits and pieces to create the final Reference Scene

```cpp
void Scene_W4_ReferenceScene::Update(Timer* pTimer)
{
    Scene::Update(pTimer);

    const auto yawAngle:float = (cos(_Xx:pTimer->GetTotal()) + 1.f) / 2.f * PI_2;
    for (const auto m:TriangleMesh* : m_Meshes)
    {
        m->RotateY(yawAngle);
        m->UpdateTransforms();
    }
}
```

```cpp
//+++++++++++++++++++++++++++++++++++++++++++
//WEEK 4 Reference Scene
class Scene_W4_ReferenceScene final : public Scene
{
public:
    Scene_W4_ReferenceScene() = default;
    ~Scene_W4_ReferenceScene() override = default;

    Scene_W4_ReferenceScene(const Scene_W4_ReferenceScene&) = delete;
    Scene_W4_ReferenceScene(Scene_W4_ReferenceScene&&) noexcept = delete;
    Scene_W4_ReferenceScene& operator=(const Scene_W4_ReferenceScene&) = delete;
    Scene_W4_ReferenceScene& operator=(Scene_W4_ReferenceScene&&) noexcept = delete;

    void Initialize() override;
    void Update(Timer* pTimer) override;

private:
    TriangleMesh* m_Meshes[3]{};
};
```

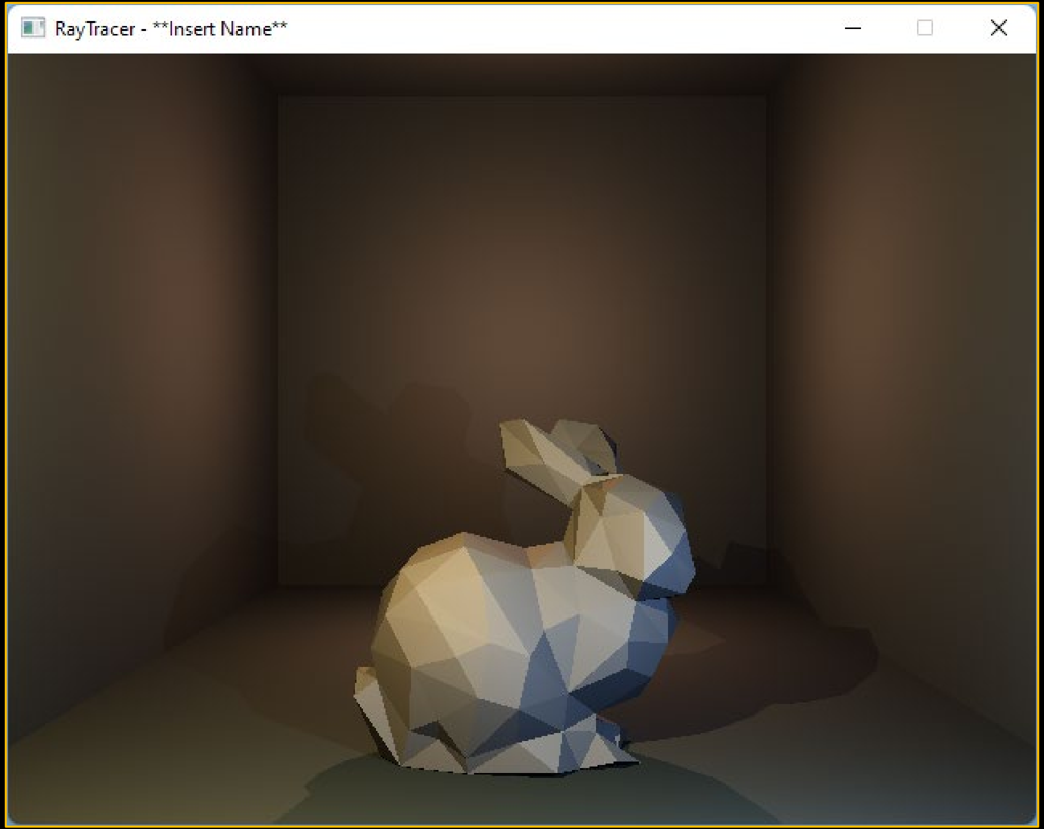
Raytracing | Reference Scene

- Now we have all the bits and pieces to create the final Reference Scene



Raytracing | Bunny Scene (12)

- Use the lowpoly_bunny2 OBJ (leho) to create a Scene_W4_BunnyScene
  - Scene will render at a very low framerate!
  - Bunny in screenshot has a uniform scale of 2



GOOD LUCK!