GRAPHICS PROGRAMMING I

OPTIMIZATIONS

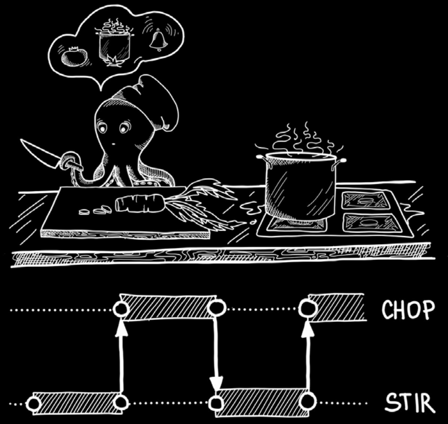
Ray Tracing| Optimizations

- Increase the intersection speed
  - More performant Hit Algorithms
    - Moller Trumbore (Triangle Intersection)
    - Sphere HitTest (Geometric vs Analytic)
    - …
- Reduce the number of intersections
  - Acceleration Structures
    - Axis-Aligned Bounding Boxes (AABB) - SlabTest
    - Uniform Grid Algorithm
    - Bounding Volume Hierarchies (BVH)
    - Octree Algorithm
    - Binary Space Partitioning (BSP) Tree Algorithm
    - KD-Tree Algorithm
    - …

- Reduce the number of rays
- Data vs Object Oriented framework
- Use parallel algorithms
  - Multithreading
    - Thread / ThreadPools
    - Parallel For
    - Async

Ray Tracing| Multithreading

A software implementation allowing different threads to be executed concurrently. A multithreaded program appears to be doing several things at the same time even when it's running on a single-core machine (concurrent).

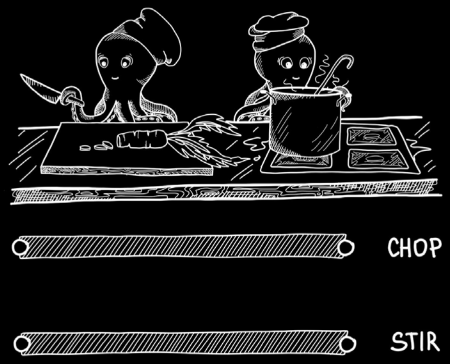


Concurrency vs Parallelism

- Concurrency
  - Executing multiple tasks at the same time but not necessarily simultaneously. On a single-core machine it's an illusion of multiple tasks running in parallel because of a very fast switching between threads/tasks by the CPU.

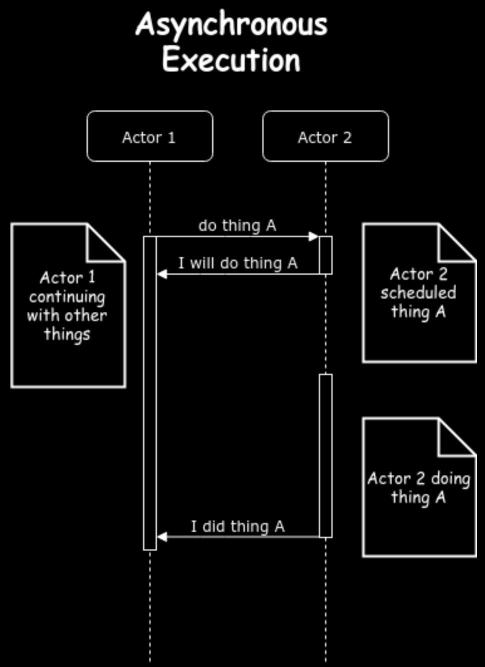
- Parallelism
  - When the tasks are actually being executed in parallel on multiple cores.

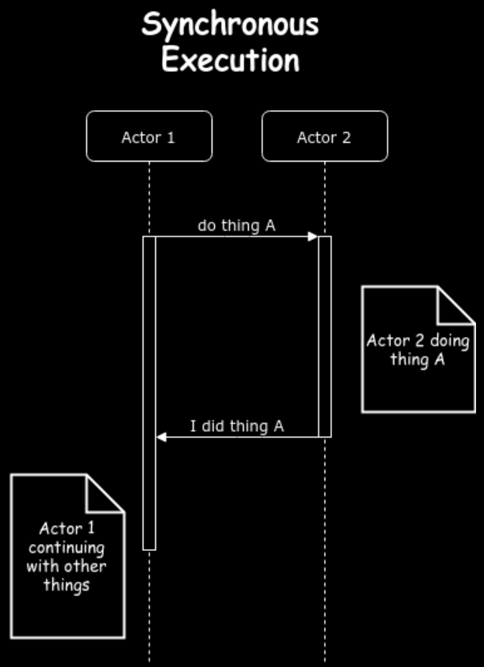
- Multi-core machines will of course use a combination of concurrency and parallelism



Ray Tracing| Async

- Asynchrony
  - Refers to the fact that one event might be happening at a different time (not in synchrony) to another event.
  - Depending on the execution context, async execution can run in parallel
    (std::async > Depends on the context, compiler & operating system)
  - Commonly used to execute tasks on a separate thread to prevent blocking the main thread (ex. UI Responsiveness)





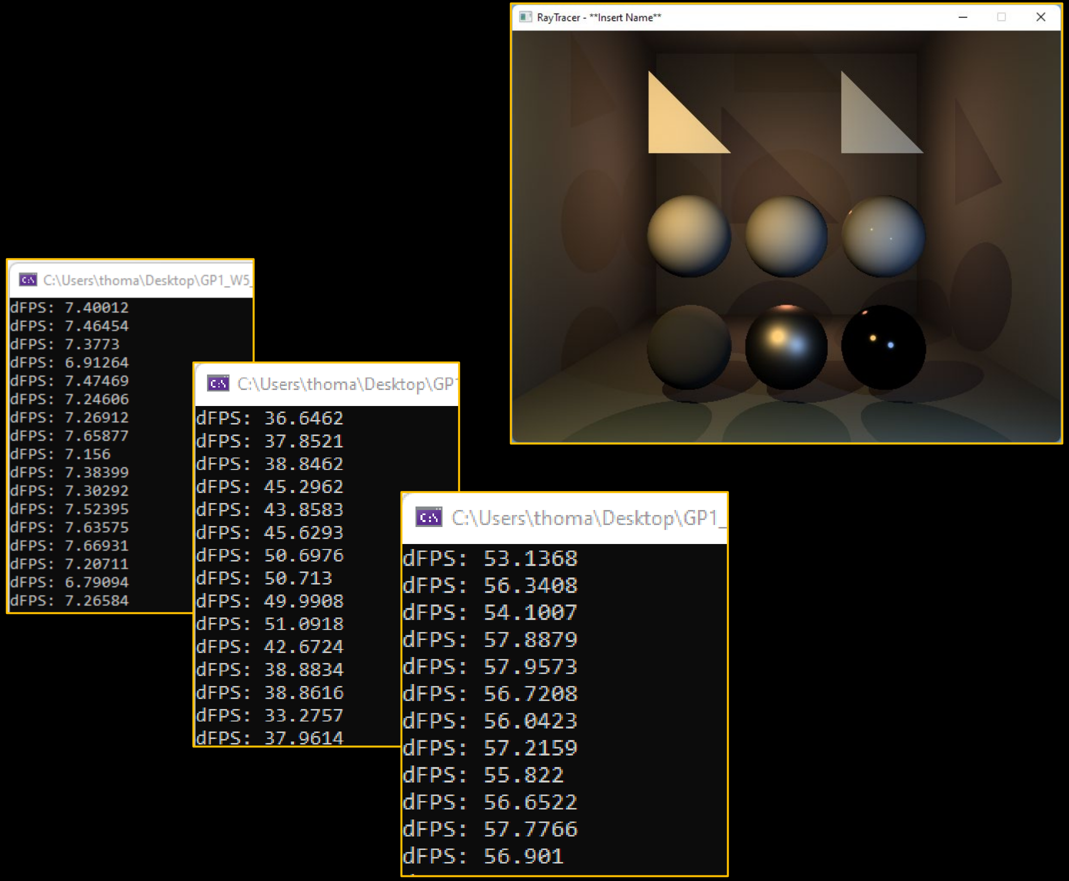
Ray Tracing| Hands-On

1. Restructure ‘renderer’ > Function to render a single pixel (task)

2. Synchronous Execution (default)

3. Asynchronous Execution (std::async)
    1. Limited to the number of logical cores (semi-parallel)

4. Parallel Execution (std::parallel_for)



Ray Tracing| Hands-On (MultiThreading)

- Create function to process a single pixel (Renderer::RenderPixel)

Renderer.h

```
void Render(Scene* pScene) const;

void RenderPixel(Scene* pScene, uint32_t pixelIndex, float fov, float aspectRatio, const Camera& camera, const std::vector<Light>& lights, const std::vector<Material*>& materials) const;

bool SaveBufferToImage() const;
```

Renderer.cpp

```
void Renderer::RenderPixel(Scene* pScene, uint32_t pixelIndex, float fov, float aspectRatio,
    const Camera& camera, const std::vector<Light>& lights, const std::vector<Material*>& materials) const
{
    const int px = pixelIndex % m_Width;
    const int py = pixelIndex / m_Width;

    float rx = px + 0.5f;
    float ry = py + 0.5f;

    float cx = (2 * (rx / float(m_Width)) - 1) * aspectRatio * fov;
    float cy = (1 - (2 * (ry / float(m_Height)))) * fov;

    //Same code you already have but without running it inside a double for-loop
    //...
```

Ray Tracing| Hands-On (MultiThreading)

- Renderer::Render
  - The majority of the code can be removed (double for-loop)
  - Data that every pixel needs for processing can be prefetched here
    - Camera (& cameraToWorld matrix calculation)
    - Materials
    - Lights
    - Aspect Ratio
    - Field of View
  - Calculate the total number of pixels (width * height)
  - Make use of 'preprocessor directives' to switch between implementations

Renderer.cpp (top)

```cpp
//#define ASYNC
//#define PARALLEL_FOR
```

Renderer.cpp

```cpp
void Renderer::Render(Scene* pScene) const
{
    Camera& camera = pScene->GetCamera();
    camera.CalculateCameraToWorld();

    const float fovAngle = camera.fovAngle * TO_RADIANS;
    const float fov = tan( _Xx: fovAngle / 2.f);

    const float aspectRatio = m_Width / static_cast<float>(m_Height);

    auto& materials :const vector<Material*>& = pScene->GetMaterials();
    auto& lights :const vector<Light>& = pScene->GetLights();

    const uint32_t numPixels = m_Width * m_Height;

#if defined(ASYNC)
    //Async Logic
#elif defined(PARALLEL_FOR)  #if defined(ASYNC)
    //Parallel-For Logic
#else  #elif defined(PARALLEL_FOR)
    //Synchronous Logic (no threading)
#endif  #elif defined(PARALLEL_FOR) #else

    //@END
    //Update SDL Surface
    SDL_UpdateWindowSurface(m_pWindow);
}
```

Ray Tracing| Hands-On (MultiThreading)

- Synchronous Logic (No Threading)
  - Simple for-loop that loops numPixels amount of times

Renderer::Render

```
#else  #elif defined(PARALLEL_FOR)

    //No Threading
    //+++++++++++++
    for (uint32_t i{ 0 }; i < numPixels; ++i)
    {
        RenderPixel(pScene, i, fov, aspectRatio, pScene->GetCamera(), lights, materials);
    }

#endif  #elif defined(PARALLEL_FOR) #else
```

Ray Tracing| Hands-On (MultiThreading)

```
#define ASYNC
//#define PARALLEL_FOR
```

- Async Logic (using futures) [1]
  - Similar to the 'synchronous' approach, but now we are spreading the work across multiple async operations
  - Systems decides how these are scheduled (could be parallel, could be concurrent) – depends on system resources
  - We are limiting the amount of async operations to the total number of cores we have available
  - Requires the 'future' header

Renderer.cpp

```cpp
#include <future> //async
```

Renderer::Render (1)

```cpp
#if defined(ASYNC)
	//Async
	//+++++
	const uint32_t numCores = std::thread::hardware_concurrency();
	std::vector<std::future<void>> async_futures{};
	const uint32_t numPixelsPerTask = numPixels / numCores;
	uint32_t numUnassignedPixels = numPixels % numCores;
	uint32_t currPixelIndex = 0;
```

Ray Tracing| Hands-On (MultiThreading)

- Async Logic (using futures) [2]

Renderer::Render (2)

```
    for(uint32_t coreId{0}; coreId < numCores; ++coreId)
    {
        uint32_t taskSize = numPixelsPerTask;
        if (numUnassignedPixels > 0)
        {
            ++taskSize;
            --numUnassignedPixels;
        }

        async_futures.push_back( _Val: std::async( _Policy: std::launch::async, _Fnarg: [=, this] ->void
            {
                //Render all pixels for this task (currPixelIndex > currPixelIndex + taskSize)
                const uint32_t pixelIndexEnd = currPixelIndex + taskSize;
                for (uint32_t pixelIndex{ currPixelIndex }; pixelIndex < pixelIndexEnd; ++pixelIndex)
                {
                    RenderPixel(pScene, pixelIndex, fov, aspectRatio, camera, lights, materials);
                }
            }));

        currPixelIndex += taskSize;
    }

    //Wait for async completion of all tasks
    for (const std::future<void>& f : async_futures)
    {
        f.wait();
    }

#elif defined(PARALLEL_FOR)  #if defined(ASYNC)
```

Ray Tracing| Hands-On (MultiThreading)

```
//#define ASYNC
#define PARALLEL_FOR
```

- Parallel Logic
  - We can easily parallelize our for-loop by making use of the “Parallel Patterns Library” (PPL) which is a part of the C++ Standard Library
  - Same as with the Async approach, the system is in charge of scheduling the tasks but ensure execution in a parallel manner depending on the available number of computing resources. (Should give you better performance compared to the async approach)
    https://learn.microsoft.com/en-us/cpp/parallel/concrt/parallel-algorithms?view=msvc-170
  - Requires the ‘ppl’ header (+ parallel functions are part of the ‘concurrency’ namespace)

Renderer.cpp

```
#include <ppl.h> //parallel_for
```

Renderer::Render

```
#elif defined(PARALLEL_FOR)  #if defined(ASYNC)

    //Parallel For
    //+++++++++++++
    concurrency::parallel_for(_First: 0u, _Last: numPixels, _Func: [=, this](int i) ->void {
        RenderPixel(pScene, i, fov, aspectRatio, camera, lights, materials);
        });

#else  #elif defined(PARALLEL_FOR)
```

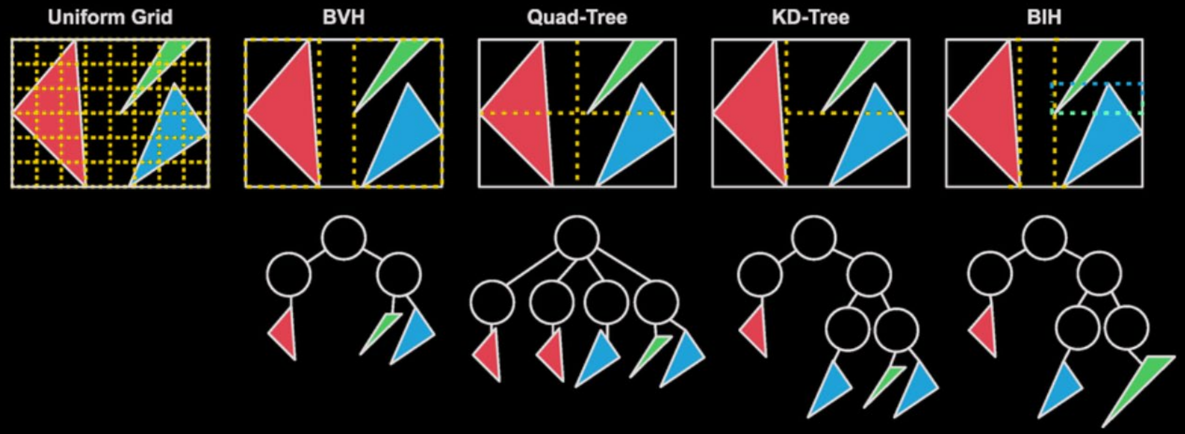
- parallel_for (concurrency namespace)
  - Works very similar to a traditional for-loop (iterating over a specified range) but with automated parallelization under the hood

Ray Tracing| Acceleration Structure

- An acceleration structure is super useful to prevent unnecessary calculations (= reducing the number of intersection test per frame)

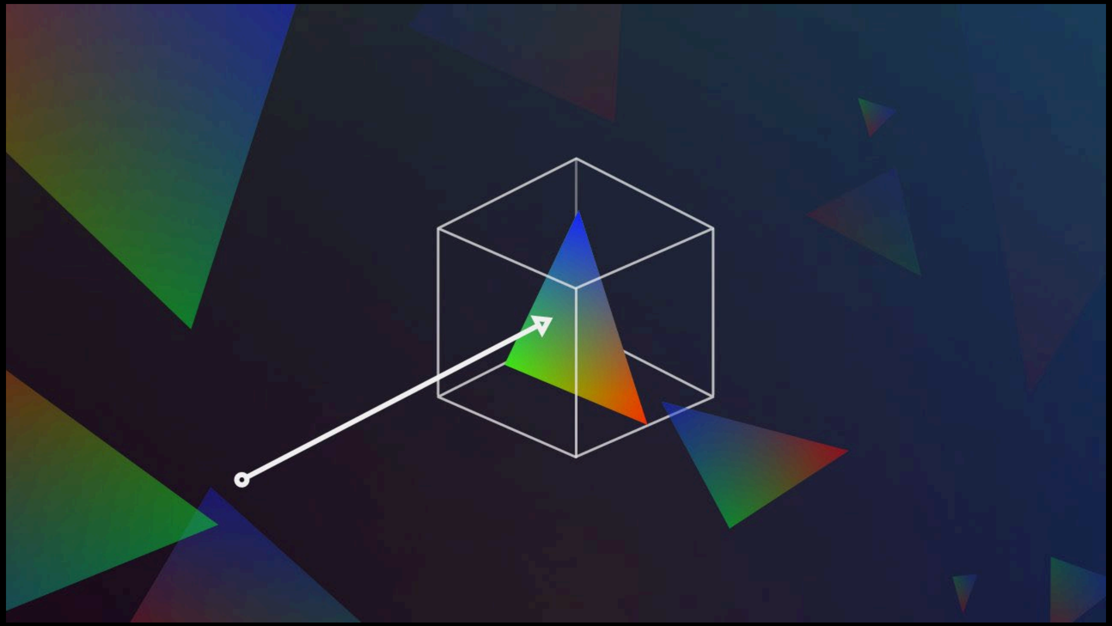
- There are lots of acceleration techniques available

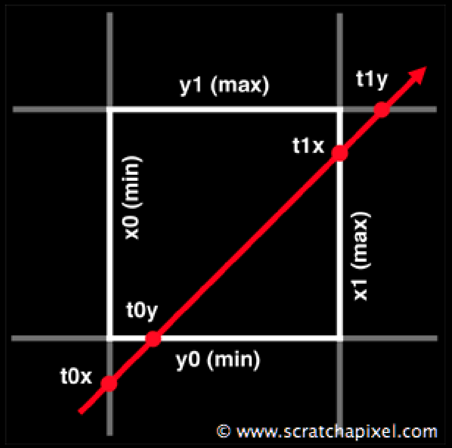
- For this hands-on we are going to implement a very simple one, The Slab-Test
  > AABB bases pre-hit check for our TriangleMeshes
  > AABB (Axis-Aligned BoundingBox)



Ray Tracing| Acceleration Structure

- Slab Test (AABB-Ray Intersection)
  1. Define a AABB around the TriangleMesh (Min/Max)
  2. Perform a HitTest with the AABB box
     1. Test each 'slab'
  3. If NO HIT > Do not HitTest the TriangleMesh
  4. IF HIT > Perform TriangleMesh HitTest





$$t0x = (B0_x - O_x)/D_x$$
$$t1x = (B1_x - O_x)/D_x$$
$$t0y = (B0_y - O_y)/D_y$$
$$t1y = (B1_y - O_y)/D_y$$
$$t0z = (B0_z - O_z)/D_z$$
$$t1z = (B1_z - O_z)/D_z$$

Ray Tracing| Acceleration Structure

- Update Vector3 class
  - Vector3::Min > Get smallest components of 2 vectors
  - Vector3::Max > Get biggest components of 2 vectors

Vector3.h

```cpp
static Vector3 Reflect(const Vector3& v1, const Vector3& v2);

static Vector3 Max(const Vector3& v1, const Vector3& v2);
static Vector3 Min(const Vector3& v1, const Vector3& v2);
```

Vector3.cpp

```cpp
Vector3 Vector3::Max(const Vector3& v1, const Vector3& v2)
{
    return {
        _x: std::max(_Left: v1.x, _Right: v2.x),
        _y: std::max(_Left: v1.y, _Right: v2.y),
        _z: std::max(_Left: v1.z, _Right: v2.z)
    };
}
```

Vector3.cpp

```cpp
Vector3 Vector3::Min(const Vector3& v1, const Vector3& v2)
{
    return {
        _x: std::min(_Left: v1.x, _Right: v2.x),
        _y: std::min(_Left: v1.y, _Right: v2.y),
        _z: std::min(_Left: v1.z, _Right: v2.z)
    };
}
```

Ray Tracing| Acceleration Structure

- Update TriangleMesh
  - Find AABB for (object-space) vertices (AABB is defined by a Min & Max position)
  - Calculate the AABB after transforming the vertices (world-space)

DataTypes.h (TriangleMesh)

```
Matrix scaleTransform{};

Vector3 minAABB;
Vector3 maxAABB;

Vector3 transformedMinAABB;
Vector3 transformedMaxAABB;

std::vector<Vector3> transformedPositions{};
```

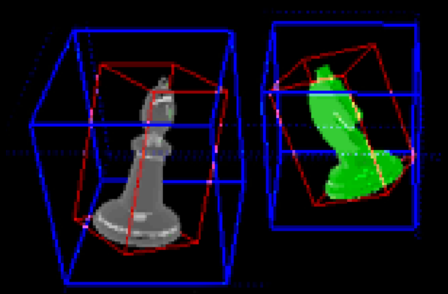
Ray Tracing| Acceleration Structure

- Update TriangleMesh
  - Function to calculate the object-space AABB (TriangleMesh::UpdateAABB)
  - Function to calculate the word-space AABB (TriangleMesh::UpdateTransformedAABB) (closest fitting AABB after transformations)

DataTypes.h (TriangleMesh)

```
void UpdateAABB()
{
    //Update AABB Logic
}

void UpdateTransformedAABB(const Matrix& finalTransform)
{
    //Update Transformed AABB
}
```



Ray Tracing| Acceleration Structure

- TriangleMesh::UpdateAABB

DataTypes.h (TriangleMesh)

```
void UpdateAABB()
{
    if (positions.size() > 0)
    {
        minAABB = positions[0];
        maxAABB = positions[0];
        for (auto& p :Vector3& : positions)
        {
            minAABB = Vector3::Min(p, minAABB);
            maxAABB = Vector3::Max(p, maxAABB);
        }
    }
}
```

Ray Tracing| Acceleration Structure

- TriangleMesh::UpdateAABB

- Make sure to Update the AABB every time the TriangleMesh::UpdateTransforms is called

TriangleMesh::UpdateTransforms

```
for (auto& p :Vector3& : positions)
{
    transformedPositions.emplace_back(
        finalTransform.TransformPoint(p));
}

// Update AABB
UpdateTransformedAABB(finalTransform);
```

DataTypes.h (TriangleMesh)

```
void UpdateTransformedAABB(const Matrix& finalTransform) {
    // AABB update: be careful -> transform the 8 vertices of the aabb
    // and calculate new min and max.
    Vector3 tMinAABB = finalTransform.TransformPoint(minAABB);
    Vector3 tMaxAABB = tMinAABB;
    // (xmax,ymin,zmin)
    Vector3 tAABB = finalTransform.TransformPoint(maxAABB.x, minAABB.y, minAABB.z);
    tMinAABB = Vector3::Min(tAABB, tMinAABB);
    tMaxAABB = Vector3::Max(tAABB, tMaxAABB);
    // (xmax,ymin,zmax)
    tAABB = finalTransform.TransformPoint(maxAABB.x, minAABB.y, maxAABB.z);
    tMinAABB = Vector3::Min(tAABB, tMinAABB);
    tMaxAABB = Vector3::Max(tAABB, tMaxAABB);
    // (xmin,ymin,zmax)
    tAABB = finalTransform.TransformPoint(minAABB.x, minAABB.y, maxAABB.z);
    tMinAABB = Vector3::Min(tAABB, tMinAABB);
    tMaxAABB = Vector3::Max(tAABB, tMaxAABB);
    // (xmin,ymax,zmin)
    tAABB = finalTransform.TransformPoint(minAABB.x, maxAABB.y, minAABB.z);
    tMinAABB = Vector3::Min(tAABB, tMinAABB);
    tMaxAABB = Vector3::Max(tAABB, tMaxAABB);
    // (xmax,ymax,zmin)
    tAABB = finalTransform.TransformPoint(maxAABB.x, maxAABB.y, minAABB.z);
    tMinAABB = Vector3::Min(tAABB, tMinAABB);
    tMaxAABB = Vector3::Max(tAABB, tMaxAABB);
    // (xmax,ymax,zmax)
    tAABB = finalTransform.TransformPoint(maxAABB);
    tMinAABB = Vector3::Min(tAABB, tMinAABB);
    tMaxAABB = Vector3::Max(tAABB, tMaxAABB);
    // (xmin,ymax,zmax)
    tAABB = finalTransform.TransformPoint(minAABB.x, maxAABB.y, minAABB.z);
    tMinAABB = Vector3::Min(tAABB, tMinAABB);
    tMaxAABB = Vector3::Max(tAABB, tMaxAABB);

    transformedMinAABB = tMinAABB;
    transformedMaxAABB = tMaxAABB;
}
```

Ray Tracing| Acceleration Structure

- Utils::SlabTest_TriangleMesh

Utils.h

```cpp
inline bool SlabTest_TriangleMesh(const TriangleMesh& mesh, const Ray& ray)
{
    float tx1 = (mesh.transformedMinAABB.x - ray.origin.x) / ray.direction.x;
    float tx2 = (mesh.transformedMaxAABB.x - ray.origin.x) / ray.direction.x;

    float tmin = std::min(tx1, tx2);
    float tmax = std::max(tx1, tx2);

    float ty1 = (mesh.transformedMinAABB.y - ray.origin.y) / ray.direction.y;
    float ty2 = (mesh.transformedMaxAABB.y - ray.origin.y) / ray.direction.y;

    tmin = std::max(tmin, std::min(ty1, ty2));
    tmax = std::min(tmax, std::max(ty1, ty2));

    float tz1 = (mesh.transformedMinAABB.z - ray.origin.z) / ray.direction.z;
    float tz2 = (mesh.transformedMaxAABB.z - ray.origin.z) / ray.direction.z;

    tmin = std::max(tmin, std::min(tz1, tz2));
    tmax = std::min(tmax, std::max(tz1, tz2));

    return tmax > 0 && tmax >= tmin;
}
```

Ray Tracing| Acceleration Structure

- Perform SlabTest before testing the entire TriangleMesh

Utils.h (HitTest_TriangleMesh)

```
inline bool HitTest_TriangleMesh(const TriangleMesh& mesh, const Ray& ray, HitRecord& hitRecord, bool ignoreHitRecord = false)
{
    // slabtest
    if (!SlabTest_TriangleMesh(mesh, ray)) {
        return false;
    }

    Triangle triangle{};
```

Ray Tracing| Acceleration Structure

- Use in your scenes!

Scene_W4_Bunny::Initialize

```
m_pMesh = AddTriangleMesh(TriangleCullMode::BackFaceCulling, matLambert_White);
Utils::ParseOBJ(filename: "Resources/lowpoly_bunny2.obj",
    [&] m_pMesh->positions,
    [&] m_pMesh->normals,
    [&] m_pMesh->indices);

m_pMesh->Scale( { _x: 2.f, _y: 2.f, _z: 2. });

m_pMesh->UpdateAABB();
m_pMesh->UpdateTransforms();
```