Document by Jesper Enschede
jesperenschede@gmail.com

This document contains code samples of a Bounding volume hierarchy (BVH) I made for a game called On The Bubble. The game is made in a custom game engine and features a custom physics engine.

I wanted to use an acceleration structure to improve the performance of our game by reducing the amount of physics objects the player's sphere needs to check for collision with. Because the game levels are made up of nearly only static geometry I opted for a BVH because it's fairly trivial to implement.

I started out by making a simple struct that represents the BVH node.

```
struct BVHNode
{
	AABB m_Box;
	unsigned m_LeftNode = 0;
	unsigned m_ObjectCount = 0;
};
```

The box is a simple axis aligned bounding box, We use this to check if we collide with the node. LeftNode is a field that has an index to the left child node, If we want to get the right child node we can do LeftNode + 1. ObjectCount is only used if this node is a leaf node. It's used to store how many physics objects are inside of this node.

LeftNode serves 2 purposes. It's either an index into an array pointing to the left child node, or if the ObjectCount is larger than 0 it is an index into an array pointing to the physics objects in this node.
The reason why we have a member variables that has 2 purposes is for optimization reasons. The AABB is six 32 bit floats, totaling 24 bytes, then we have two 4 byte integers. Meaning our Node struct is 32 bytes total. Moderns PCs usually have a cache line of 64 bytes. Meaning if we use an aligned allocator we can have 2 node structs per cache line.

In order to build the BVH we have a simple function that takes in a reference to a std::vector of physics objects

```
void build(std::vector<class PhysicsObject *> &physicsObjects);
```

The physics engine stores them in a std::vector, So I went along with it. The reason I used a separate build function rather than a constructor is because we might not want to build the BVH immediately whenever we allocate the BVH object in our engine.

```
m_NodePoolSize = amountOfObjects * 2;
m_NodePool = _alligned_malloc(sizeof(BVHNode) * m_NodePoolSize, 64);
```
We pre-allocate a pool of nodes. We do this for speed reasons. We know how many nodes we need in the worst case scenario. Allocating them in an array also means they are next to each other in memory, In theory reducing cache misses when traversing the BVH. We used aligned malloc to ensure that we have two BVH nodes per cache line.

```
m_RootNode = &m_NodePool[0];
m_RootNode->m_LeftNode = 0;
m_RootNode->m_ObjectCount = physicsObjects.size();
```
We assign some default values to the root node. We use this node to further subdivide the physics objects into nodes.

We then set the bounds on the root node. In order to get the bounds we simply loop over all the physics objects in the node. Then we use a min and max function to calculate the bounds.

```
void BVH::SetNodeBounds(BVHNode &node)
{
	const int BigValue = 1'000'000'000;
	glm::vec3 min = glm::vec3(BigValue, BigValue, BigValue);
	glm::vec3 max = glm::vec3(-BigValue, -BigValue, -BigValue);

	unsigned first = node.m_LeftNode;
	for (size_t i = 0; i < node.m_ObjectCount; i++)
	{
		int objectIndex = m_ObjectsIndices[first + i];
		PhysicsObject *object = m_Objects[objectIndex];
		Shape *shape = object->GetShape();

		const AABB &objectAabb = shape->GetAABB();

		min = glm::min(min, objectAabb.m_Minimum);
		max = glm::max(max, objectAabb.m_Maximum);
	}

	node.box.m_Minimum = min;
	node.box.m_Maximum = max;
```

}

After we calculate the bounds of the root note we call a subdivide function. This function calculates a split point and sorts the physics objects based on this split point. Then it recursively calls the Subdivide function on the child nodes in order to create a BVH.

In order to calculate the split point we check which axis of the box extend (a vector from the max to min) is the longest, we then split that axis down the middle. I could've used a surface area heuristic (SAH) technique in order to find a more optimal split axis. However, to keep the BVH construction simple I opted for a simple 'longest axis split' initially to see if it was fine for our current needs in the project. And to this day it works fine. I might add SAH if needed later down the line.

After we calculate the split axis we sort objects based on where they fall on the split axis. We sort an array with indices to the physics objects, the reason for this is because the BVH is not owning over the memory of the physics objects.

After we are done sorting the physics objects based on the split axis we check if this node can be subdivided even further, if so, we calculate the bounds of the child nodes and we recursively call the subdivide function.

```
void BVH::SubdivideNode(BVHNode &node)
{
	if (node.m_ObjectCount <= m_MaxNodesInNonLeafNode)
	{
		return;
	}

	glm::vec3 aabbExtend = node.box.m_Maximum - node.box.m_Minimum;
	int axis = 0;

	if (aabbExtend.y > aabbExtend.x)
	{
		axis = 1;
	}
	if (aabbExtend.z > aabbExtend[axis])
	{
		axis = 2;
	}

	float splitPosition = node.box.m_Minimum[axis] + aabbExtend[axis] * 0.5f;

	int i = node.m_LeftNode;
	int j = i + node.m_ObjectCount - 1;

	while (i <= j)
```

```
        {
                auto &pos =m_Objects[m_ObjectsIndices[i]]->m_Owner->GetTransform().GetPosition();
                if (pos[axis] < splitPosition)
                {
                        i++;
                }
                else
                {
                        std::swap(m_ObjectsIndices[i], m_ObjectsIndices[j--]);
                }
        }

        int leftCount = i - node.m_LeftNode;

        if (leftCount == 0 || leftCount == node.m_ObjectCount)
        {
                return;
        }

        int leftNodeIndex = m_NodesInUse++;
        int rightNodeIndex = m_NodesInUse++;

        m_NodePool[leftNodeIndex].m_LeftNode = node.m_LeftNode;
        m_NodePool[leftNodeIndex].m_ObjectCount = leftCount;

        m_NodePool[rightNodeIndex].m_LeftNode = i;
        m_NodePool[rightNodeIndex].m_ObjectCount = node.m_ObjectCount - leftCount;

        node.m_LeftNode = leftNodeIndex;
        node.m_ObjectCount = 0;

        SetNodeBounds(m_NodePool[leftNodeIndex]);
        SetNodeBounds(m_NodePool[rightNodeIndex]);

        SubdivideNode(m_NodePool[leftNodeIndex]);
        SubdivideNode(m_NodePool[rightNodeIndex]);
}
```

BVH traversal on the next page.

In order to traverse the BVH we have a simple function that returns a std::vector with pointers to all the physics objects the player can collide with.

We check if the AABB of the player collides with the AABB of a node. If so, we check if the node has an object count higher than 0, this means that it's a leaf node and contains physics objects. If not we recursively call the traverse function on the child nodes.

In order to prevent clipping we scale the players AABB with its current velocity.

Personally I am not 100% happy with this function as of yet. Because when we traverse child nodes we get a std::vector returned and we slowly push back all of this data into the result vector.
We can probably find out what the maximum number of physics objects is that we can collide with and preallocate a std::vector or array of physics object pointers.

```
std::vector<PhysicsObject *> BVH::Traverse(PhysicsObject *physicsObject, BVHNode &node) {
	std::vector<PhysicsObject *> result;

	AABB aabb = physicsObject->GetShape()->GetAABB();

	glm::vec3 pos = physicsObject->m_Position;

	const float mag = glm::length(physicsObject->m_Velocity) *
GetLevel().GetPhysicsScene()->GetPhysicsTimeStep();

	AABB aabbInWorldSpace = {
		{pos.x + aabb.m_Minimum.x - mag, pos.y + aabb.m_Minimum.y - mag, pos.z + aabb.m_Minimum.z -
mag},
		{pos.x + aabb.m_Maximum.x + mag, pos.y + aabb.m_Maximum.y + mag, pos.z + aabb.m_Maximum.z +
mag}};

	if (Physics::DoesAABBCollide(aabbInWorldSpace, node.box))
	{
		if (node.m_ObjectCount > 0)
		{
			result.reserve(node.m_ObjectCount);

			for (size_t i = 0; i < node.m_ObjectCount; i++)
			{
				result.push_back(m_Objects[m_ObjectsIndices[node.m_LeftNode + i]]);
			}

			return result;
		}
		else
		{
			auto a = Traverse(physicsObject, m_NodePool[node.m_LeftNode]);

			for (size_t i = 0; i < a.size(); i++)
			{
				result.push_back(a[i]);
			}

			auto b = Traverse(physicsObject, m_NodePool[node.m_LeftNode + 1]);
```

```
                        for (size_t i = 0; i < b.size(); i++)
                        {
                                result.push_back(b[i]);
                        }
                }
        }
        return result;
}
```

This method was taught to me by one of our professors. He learned us this method during a semester where we learned how to make a raytracer. I changed this approach to make it work with a custom physics engine.

Currently it's for a top level acceleration structure (TLAS), it encapsulates all of the physics objects in a level. I want to extend it so it works as well as a bottom level acceleration structure (BLAS) this BLAS would encapsulate all of the triangles in a mesh.

Using this tlas for physics greatly improves our performance.

A scene with 101 cubes, 12 triangles each. Goes from around 10ms per physics world update using a brute force approach to around 0.1ms per physics world update using the BVH.

Thank you for taking the time to read my explanation of the BVH I made for On the bubble.
If you have any questions feel free to email me at: jesperenschede@gmail.com