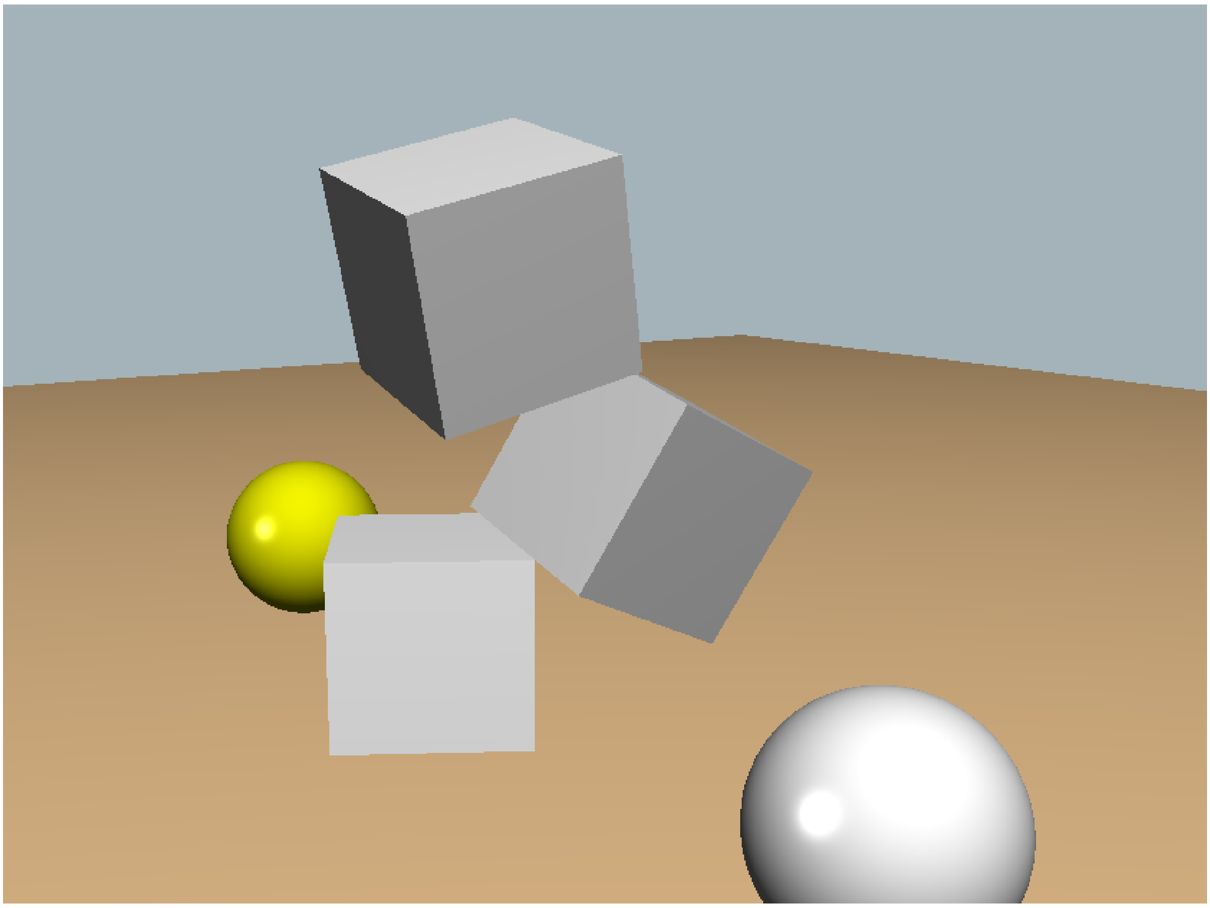
CS550
Process Paper

3D Physics Engine using Sequential Impulses

Nahye Park



Overview

The purpose of this project is building the physics engine that enables interaction between various convex-shaped objects, including cuboids, spheres, pyramids, and prisms, through accurate collision detection and resolution techniques. The dynamic AABB tree is used for the broad phase collision detection, SAT algorithms are used for the narrow phase collision detection. Sequential impulses method is implemented for the collision resolution.

Libraries

I used the OpenGL library for rendering the objects, and for mathematical computations involving vectors, quaternions, and operations like dot product and cross product, I utilized the glm headers. ImGui is used for the UI in the scene.

Physics Engine Architecture

The physics engine comprises two main components: Rigidbody and Collider. The Collider component includes four types of colliders, sphere, AABB, OBB, and convex collider, with AABB being predominantly used for the dynamic AABB tree, which means that every collider already has an AABB collider. In the physics system, there are three main functions: Integrate, Collision Detection, and Collision Resolution. The Collision Detection system encompasses both broad phase and narrow phase collision detection systems.

Physics Engine Flow

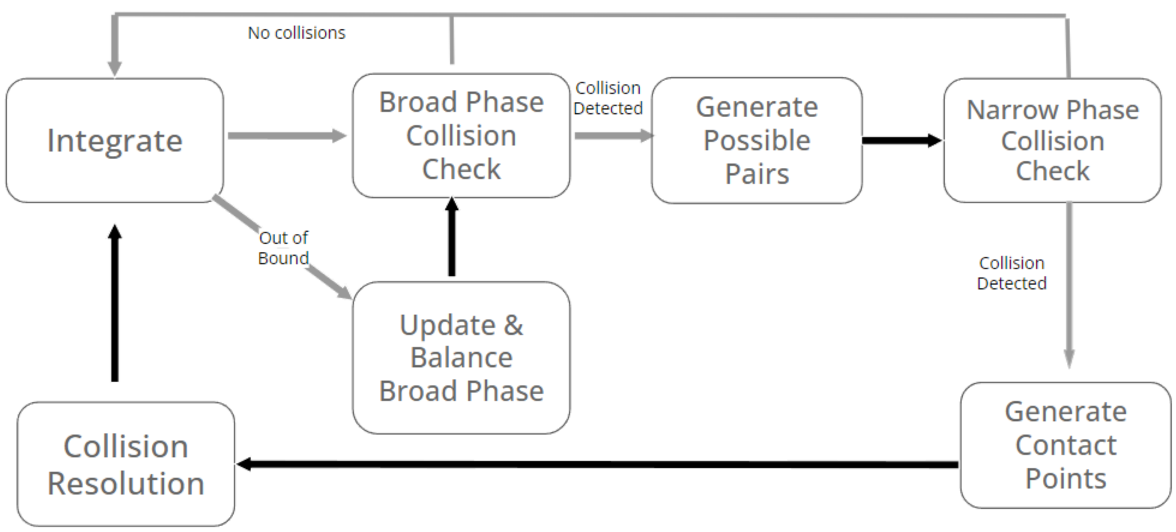


Figure 1. The diagram of physics engine flow

The Integrate function updates the objects' status using linear and angular velocities, and then updates the colliders using the objects' updated position and rotation. Additionally, it updates the dynamic AABB tree. During Collision Detection, the broad phase collision is checked first, followed by the narrow phase collision detection only for the possible set of objects. After the collision detection, contact manifolds are generated and used for the Collision Resolution.

Integrator

I used the semi-implicit Euler method as the integrator since it is superior to the standard Euler method. In the Integrate function, it is applied like below:

$$v(t+1) = v(t) + NetForce * InverseMass * dt$$
$$x(t+1) = x(t) + v(t+1) * dt$$

Fixed time step is implemented to ensure a stable update of the physics engine. It is limited to 60 frames per second, so the engine only updates when the delta time is less than the frame limit, as shown below:

```
const float FRAME_LIMIT = 1.f/60.f;
if(dt <= FRAME_LIMIT)
      PhysicsUpdate(dt);
```

Rigid Body Implementation

The RigidBody class comprises several member variables.

Private variables in RigidBody class (RigidBody.h)

```
glm::vec3 m_centerOfMass;
glm::vec3 m_velocity;
glm::vec3 m_angularVelocity;
glm::vec3 m_gravityForce;
glm::vec3 m_netTorque;
glm::vec3 m_netForce;
glm::mat4 m_inverseInertiaTensor;
float m_bounciness;
float m_inverseMass;
bool m_gravity;
bool m_isDynamic;
```

The 'm_isDynamic' variable is utilized to differentiate between the dynamic objects and static objects. The variables such as center of mass, inverse inertia tensor, and bounciness(restitution) are used for collision resolution, sequential impulses method. For the inertia tensor, since I used only spheres and cubes in this simulation, I put the values following:

Cube

$$\begin{bmatrix} \frac{2}{3}Ms^2 & -\frac{1}{4}Ms^2 & -\frac{1}{4}Ms^2 \\ -\frac{1}{4}Ms^2 & \frac{2}{3}Ms^2 & -\frac{1}{4}Ms^2 \\ -\frac{1}{4}Ms^2 & -\frac{1}{4}Ms^2 & \frac{2}{3}Ms^2 \end{bmatrix}$$

, where M is mass of the cube and s is radius of the cube.

Sphere

$$\begin{bmatrix} \frac{2}{5}Mr^2 & 0 & 0 \\ 0 & \frac{2}{5}Mr^2 & 0 \\ 0 & 0 & \frac{2}{5}Mr^2 \end{bmatrix}$$

, where M is mass of the sphere and r is radius of the sphere.

Additionally, there is transform information, including position, rotation (as a quaternion), and scale, which are also part of the object class and utilized for rendering. The majority of the functions in the RigidBody class are getters and setters, but it also includes functions that add force and torque, and apply gravity force. The update for these forces and torques is implemented in the Integrate function within the Physics.cpp file.

Functions in RigidBody class (RigidBody.cpp)

```
void RigidBody::AddForce(const glm::vec3& force)
{
	m_netForce += force;
}
void RigidBody::AddForceL(float x, float y, float z)
{
	m_netForce += glm::vec3(x,y,z);
}
void RigidBody::AddTorque(const glm::vec3& force)
{
	m_netTorque += force;
}
void RigidBody::ApplyGravity()
{
	AddForce(m_gravityForce * m_inverseMass);
}
```

Broad Phase Collision Detection

I used Erin Catto's dynamic AABB tree for the broad phase collision detection. When the program begins or a new object is generated, the AABB colliders of all objects in the scene are inserted as a leaf node. During each update frame, the tree is updated for dynamic objects as well. The leaf nodes have larger AABB boxes, so the tree verifies if the object is outside of the larger AABB box. If the larger AABB box is unable to contain the object, the leaf is removed and reinserted. During each insertion, the tree balances itself using the surface area heuristic, which is the perimeter of the box. To check for collisions in the dynamic AABB tree, it traverses the tree from the root node and examines whether two child AABB boxes intersect. If they do intersect, it checks the narrow phase collision detection between the objects inside the nodes; otherwise, it proceeds to the child nodes.

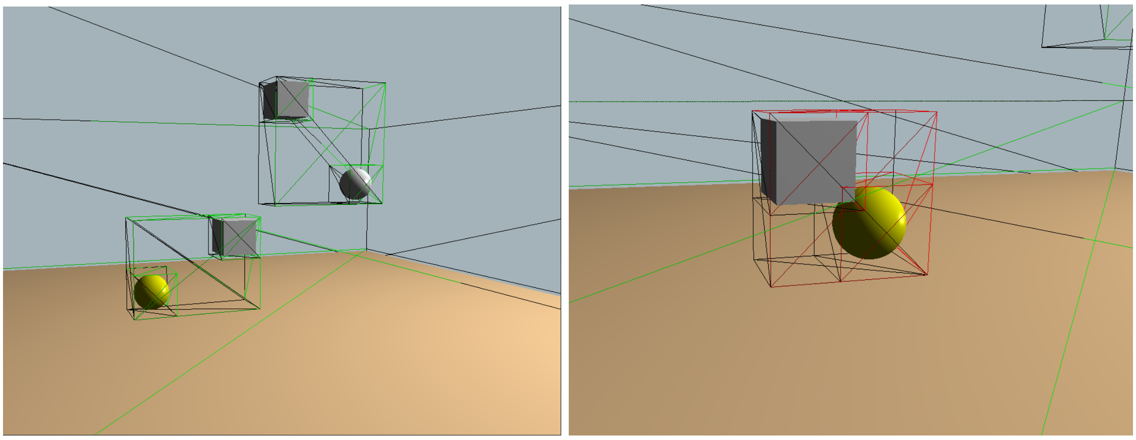


Figure 2. The changes of the dynamic AABB tree in run time.

Node struct (BVH.h)

```
struct Node
{
   AABB m_box;
   RigidBody* m_clientData;

   int m_left = -1;
   int m_right = -1;
   int m_parent;
   int m_hieght;

   bool IsLeaf() { return m_left == -1 && m_right == -1; }
};
```

Member variables in AABBDynamicTree class (BVH.h)

```
class AABBDynamictreee
{
    …
     std::vector<std::shared_ptr<Node>> nodes;
     int rootIndex = -1;
     std::queue<int> freeList;

   …
}
```

Helper functions in AABBDynamicTree class (BVH.h)

```
AABB Union(const AABB& a, const AABB& b)
{
   AABB c;
   c.m_shape = aabbBox;
   c.lower.x = min(min(a.lower.x, b.lower.x), min(a.upper.x, b.upper.x));
   c.lower.y = min(min(a.lower.y, b.lower.y), min(a.upper.y, b.m_upper.y));
   c.lower.z = min(min(a.lower.z, b.lower.z), min(a.upper.z, b.m_upper.z));
   c.upper.x = max(max(a.lower.x, b.lower.x), max(a.upper.x, b.m_upper.x));
   c.upper.y = max(max(a.lower.y, b.lower.y), max(a.upper.y, b.m_upper.y));
```

```
    c.upper.z = max(max(a.lower.z, b.lower.z), max(a.upper.z, b.m_upper.z));
    return c;
}

float Area(const AABB& a)
{
    glm::vec3 diff = a.m_upper - a.m_lower;
    return (diff.x + diff.y + diff.z) * 2.0f;
}
```

Insert function in AABBDynamicTree class (BVH.cpp)

```
void AABBDynamicTree::Insert(RigidBody* data)
{
    //Allocate new node and initialize it with data
    int leaf = AllocateNode();

    nodes[leaf]->m_box.m_shape = aabbBox;
    nodes[leaf]->m_box.m_lower = data->m_collider->m_aabb.m_lower - extent;
    nodes[leaf]->m_box.m_upper = data->m_collider->m_aabb.m_upper + extent;
    nodes[leaf]->m_left = -1;
    nodes[leaf]->m_right = -1;
    nodes[leaf]->m_clientData = data;
    nodes[leaf]->m_hieght = 0;

    //If the tree is empty
    if (rootIndex == -1) {
        rootIndex = leaf;
        nodes[rootIndex]->m_parent = -1;
        return;
     }

    //1 : Find best sibling
    AABB leafAABB = nodes[leaf]->m_box;
    int sibling = rootIndex;
    while (!nodes[sibling]->IsLeaf()) {
        int left = nodes[sibling]->m_left;
        int right = nodes[sibling]->m_right;

        AABB combined = Union(nodes[sibling]->m_box, leafAABB);
        float combinedArea = Area(combined);
        float cost = 2.0f * combinedArea;
        float inheritCost = 2.0f*(combinedArea - Area(nodes[sibling]->m_box));

        float leftCost = inheritCost;
        if (nodes[left]->IsLeaf())
            leftCost += Area(Union(nodes[left]->m_box, leafAABB));
        else
            leftCost += Area(Union(nodes[left]->m_box, leafAABB))
                        - Area(nodes[left]->m_box);

        float rightCost = inheritCost;
        if (nodes[right]->IsLeaf())
```

```
            rightCost += Area(Union(nodes[right]->m_box, leafAABB));
        else
            rightCost += Area(Union(nodes[right]->m_box, leafAABB))
                         - Area(nodes[right]->m_box);

        if (cost < leftCost && cost < rightCost) //minimum cost
            break;

        if (leftCost < rightCost)
            sibling = left;
        else
            sibling = right;
    }

    //2 : Create a new parent
    int oldParent = nodes[sibling]->m_parent;

    int newParent = AllocateNode();
    nodes[newParent]->m_parent = oldParent;
    nodes[newParent]->m_clientData = nullptr;
    nodes[newParent]->m_box = Union(leafAABB, nodes[sibling]->m_box);
    nodes[newParent]->m_hieght = nodes[sibling]->m_hieght + 1;

    if (oldParent != -1) {
        if (nodes[oldParent]->m_left == sibling)
            nodes[oldParent]->m_left = newParent;
        else
            nodes[oldParent]->m_right = newParent;

        nodes[newParent]->m_left = sibling;
        nodes[newParent]->m_right = leaf;
        nodes[sibling]->m_parent = newParent;
        nodes[leaf]->m_parent = newParent;
    }
    else {
        nodes[newParent]->m_left = sibling;
        nodes[newParent]->m_right = leaf;
        nodes[sibling]->m_parent = newParent;
        nodes[leaf]->m_parent = newParent;
        rootIndex = newParent;
    }

    //3 : Walk back up the tree refitting AABBs
    int index = nodes[leaf]->m_parent;
    while (index != -1) {
        index = Balance(index);
        int left = nodes[index]->m_left;
        int right = nodes[index]->m_right;

        nodes[index]->m_hieght = max(nodes[left]->m_hieght,
                                     nodes[right]->m_hieght) + 1;
        nodes[index]->m_box = Union(nodes[left]->m_box, nodes[right]->m_box);
        index = nodes[index]->m_parent;
    }
}
```

Remove function in AABBDynamicTree class (BVH.cpp)

```
void AABBDynamicTree::Remove(int index)
{
	if (index == rootIndex) {
		rootIndex = -1;
		FreeNode(index);
		return;
	}

	int parent = nodes[index]->m_parent;
	int grandParent = nodes[parent]->m_parent;
	int sibling;
	if (nodes[parent]->m_left == index)
		sibling = nodes[parent]->m_right;
	else
		sibling = nodes[parent]->m_left;


	if (grandParent == -1) {
		rootIndex = sibling;
		nodes[sibling]->m_parent = -1;
		FreeNode(parent);
	}
	else {
		if (nodes[grandParent]->m_left == parent)
			nodes[grandParent]->m_left = sibling;
		else
			nodes[grandParent]->m_right = sibling;

		nodes[sibling]->m_parent = grandParent;
		FreeNode(parent);

		int i = grandParent;
		while (i != -1) {
			i = Balance(i);
			int left = nodes[i]->m_left;
			int right = nodes[i]->m_right;

			nodes[i]->m_box = Union(nodes[left]->m_box,
			                        nodes[right]->m_box);
			nodes[i]->m_hieght = 1 + std::max(nodes[left]->m_hieght,
			                                  nodes[right]->m_hieght);
			i = nodes[i]->m_parent;
		}
	}
	FreeNode(index);
}
```

Balance function in AABBDynamicTree class (BVH.cpp)

```
int AABBDynamicTree::Balance(int index)
{


	//          A
	//       /      \
	//      B        C
	//    /   \    /   \
	//   D     E  F     G


	std::shared_ptr<Node> A = nodes[index];
	if (A->IsLeaf() || A->m_hieght < 2)
		return index;

	std::shared_ptr<Node> B = nodes[A->m_left];
	std::shared_ptr<Node> C = nodes[A->m_right];

	int balance = C->m_hieght - B->m_hieght;
	int a = index;
	int b = A->m_left;
	int c = A->m_right;

	// If right subtree height is bigger, rotate C up
	if (balance > 1) {
		std::shared_ptr<Node> F = nodes[C->m_left];
		std::shared_ptr<Node> G = nodes[C->m_right];
		int f = C->m_left;
		int g = C->m_right;

		// A <-> C
		C->m_left = a;
		C->m_parent = A->m_parent;
		A->m_parent = c;

		//Correct the parent's children
		if (C->m_parent == -1)
			rootIndex = c;
		else {
			if (nodes[C->m_parent]->m_left == a)
				nodes[C->m_parent]->m_left = c;
			else
				nodes[C->m_parent]->m_right = c;
		}

		//Rotate
		if (F->m_hieght > G->m_hieght) {
			C->m_right = f;
			A->m_right = g;
			G->m_parent = a;
			A->m_box = Union(B->m_box, G->m_box);
			C->m_box = Union(A->m_box, F->m_box);
```

```
            A->m_hieght = std::max(B->m_hieght, G->m_hieght) + 1;
            C->m_hieght = std::max(A->m_hieght, F->m_hieght) + 1;
        }
        else {
            C->m_right = g;
            A->m_right = f;
            F->m_parent = a;
            A->m_box = Union(B->m_box, F->m_box);
            C->m_box = Union(A->m_box, G->m_box);

            A->m_hieght = std::max(B->m_hieght, F->m_hieght) + 1;
            C->m_hieght = std::max(A->m_hieght, G->m_hieght) + 1;
        }

        return c;
    }

    // If left subtree height is bigger, rotate B up
    if (balance < -1) {
        int d = B->m_left;
        int e = B->m_right;
        std::shared_ptr<Node> D = nodes[d];
        std::shared_ptr<Node> E = nodes[e];

        // A <-> B
        B->m_left = a;
        B->m_parent = A->m_parent;
        A->m_parent = b;

        if (B->m_parent == -1)
            rootIndex = b;
        else {
            if (nodes[B->m_parent]->m_left == a)
                nodes[B->m_parent]->m_left = b;
            else
                nodes[B->m_parent]->m_right = b;
        }
        //Rotate
        if (D->m_hieght > E->m_hieght) {
            B->m_right = d;
            A->m_left = e;
            E->m_parent = a;
            A->m_box = Union(C->m_box, E->m_box);
            B->m_box = Union(A->m_box, D->m_box);

            A->m_hieght = std::max(C->m_hieght, E->m_hieght) + 1;
            B->m_hieght = std::max(A->m_hieght, D->m_hieght) + 1;
        }
        else {
            B->m_right = e;
            A->m_left = d;
            D->m_parent = a;
            A->m_box = Union(C->m_box, D->m_box);
            B->m_box = Union(A->m_box, E->m_box);
```

```
                A->m_hieght = std::max(C->m_hieght, D->m_hieght) + 1;
                B->m_hieght = std::max(A->m_hieght, E->m_hieght) + 1;
            }
            return b;
      }

      return a;
}
```

Update function in AABBDynamicTree class (BVH.cpp)

```
void AABBDynamicTree::Update()
{
   std::queue<int> q;
   std::vector<std::pair<int, RigidBody*>> datas;

   if (rootIndex != -1)
      q.push(rootIndex);

   while (!q.empty()) {
      int currIndex = q.front();
      q.pop();

      if (nodes[currIndex]->m_left != -1)
          q.push(nodes[currIndex]->m_left);
      if (nodes[currIndex]->m_right != -1)
          q.push(nodes[currIndex]->m_right);


       AABB exactAABB = nodes[currIndex]->m_clientData->m_collider->m_aabb;

       //Check if the object is outside of larger AABB box
       if (nodes[currIndex]->IsLeaf()
            && !nodes[currIndex]->m_box.Contains(exactAABB))
          datas.emplace_back(std::make_pair(currIndex,

nodes[currIndex]->m_clientData));
   }
   for (int i = 0; i < datas.size(); ++i)
      Remove(datas[i].first);

   for (int i = 0; i < datas.size(); ++i)
      Insert(datas[i].second);
}
```

Collision detection using AABBDynamicTree (Physics.cpp)

```
void Physics::DetectCollisionInTree(int parent)
{
      std::queue<int> q;
      if (parent != -1)
            q.push(parent);
```

```
	std::vector<int> toCheck;
	while (!q.empty()) {
		int curr = q.front();
		q.pop();

		if (tree->nodes[curr]->m_left != -1)
			q.push(tree->nodes[curr]->m_left);
		if(tree->nodes[curr]->m_right != -1)
			q.push(tree->nodes[curr]->m_right);

		if (tree->nodes[curr]->IsLeaf())
			toCheck.emplace_back(curr);
	}

	for (unsigned int i = 0; i < toCheck.size(); ++i) {
		for (unsigned int j = i+1; j < toCheck.size(); ++j) {
			if (i == j) continue;

			 //Check if the two aabb boxes are colliding
			 //before checking the narrow phase
			if (!intersectAABB(tree->nodes[toCheck[i]]->m_box,
					   tree->nodes[toCheck[j]]->m_box))
				continue;

			RigidBody* rbA = tree->nodes[toCheck[i]]->m_clientData;
			RigidBody* rbB = tree->nodes[toCheck[j]]->m_clientData;
			if (!Colliding(rbA, rbB)) //Narrow phase collision detection
				continue;


			//Set for debug drawing
			tree->nodes[toCheck[i]]->m_box.isColliding = true;
			tree->nodes[toCheck[j]]->m_box.isColliding = true;
			rbA->m_collider->m_color = glm::vec3(1, 0, 0);
			rbB->m_collider->m_color = glm::vec3(1, 0, 0);
		}
	}
}

bool Physics::Colliding(RigidBody* rbA, RigidBody* rbB)
{
	CollisionData col;
	if (intersect(rbA, rbB, col)) {
		m_CollisionQueue.push(col);
		return true;
	}

	return false;
}

void Physics::DetectCollisions(float dt)
{
	//tree traversal
	std::queue<int> q;
```

```
	if (tree->rootIndex != -1)
		q.push(tree->rootIndex);

	while (!q.empty()) {
		int curr = q.front();
		q.pop();

		int left = tree->nodes[curr]->m_left;
		int right = tree->nodes[curr]->m_right;

		if (left == -1 || right == -1)
			continue;

		//if left child and right child intersects
		 if (intersectAABB(tree->nodes[left]->m_box,
		                   tree->nodes[right]->m_box)) {
			//check narrow phase for containing nodes
			DetectCollisionInTree(curr);
			tree->nodes[curr]->m_box.isColliding = true;
		}
		else {//if no intersecting, continue traverse to child nodes
			tree->nodes[curr]->m_box.isColliding = false;
			q.push(left);
			q.push(right);
		}
	}
}
```

Narrow Phase Collision Detection

SAT works by checking if there is a separating axis between two shapes, using a list of axes. Since the simulation includes 3D polygons, I tested it by checking the normal of each face in two objects. This involves checking the face normals of Object A and Object B. In cases where one of the objects is a sphere, only the face normals of the polygon object are checked against the sphere. When detecting a collision, the normals of minimum penetration depth are stored and used to generate contact manifolds, also known as separating normals. These contact manifolds are calculated using the clipping method and the separating normal.

SAT Implementation (SAT.h)

```
static bool SATSphereConvex(RigidBody* a, RigidBody* b,
                            std::vector<ContactPoint>& cp)
{
	auto convex = std::static_pointer_cast<ConvexCollider>(a->m_collider);
	auto sphere = std::static_pointer_cast<SphereCollider>(b->m_collider);

	glm::vec3 sepAxis;
	float minDepth = FLT_MAX;
	std::pair<std::vector<size_t>,glm::vec3> face;
	for (size_t i = 0; i < convex->m_faces.size(); ++i)
```

```
	{
		glm::vec3 faceNormal = convex->m_faces[i].second;

		glm::vec3 sphereToFace =

convex->m_vertices[convex->m_faces[i].first[0]]
					   - sphere->m_position;
		float depth = glm::dot(sphereToFace, faceNormal)
				      + sphere->m_radius;

		// Found separating axis
		if (depth <= 0.f)
			return false;

		if (depth < minDepth)
		{
			face = { convex->m_faces[i].first, faceNormal };
			minDepth = depth;
			sepAxis = faceNormal;
		}
	}

	//Generate contact manifold - only one for sphere vs polygon
	ContactPoint c;
	c.contactNormal = face.second;
	c.contactPointA = sphere->m_position - face.second * sphere->m_radius;
	c.contactPointB = sphere->m_position + face.second
						      * (minDepth -
sphere->m_radius);
	c.penetrationDepth = minDepth;
	cp.push_back(c);

	return true;
}

static bool SATFacePolygonGlobal(std::shared_ptr<ConvexCollider> a,
std::shared_ptr<ConvexCollider> b,
	const std::pair<std::vector<size_t>, glm::vec3>& face, float& depth)
{
	glm::vec3 support;
	b->ConvexFindFurthestPoint(-face.second, support);

	glm::vec3 faceVert = a->m_vertices[face.first[0]];
	depth = glm::dot((faceVert - support), face.second);

	if (depth <= 0.f)
		return false;

	return true;
}

static bool FindSparatingAxis(RigidBody* a, RigidBody* b, bool& flip,
						     std::vector<ContactPoint>& cp)
{
	std::shared_ptr<ConvexCollider> colA =
```

```
std::static_pointer_cast<ConvexCollider>(a->m_collider);
	std::shared_ptr<ConvexCollider> colB =

std::static_pointer_cast<ConvexCollider>(b->m_collider);

	glm::vec3 diff = colA->m_position - colB->m_position;
	std::vector<size_t> faceA;
	std::vector<size_t> faceB;
	flip = false;

	float minDepthA = FLT_MAX;
	float minDepthB = FLT_MAX;

	glm::vec3 sepAxisA;
	glm::vec3 sepAxisB;

	//For each face normal of collider A
	for (size_t i = 0; i < colA->m_faces.size(); ++i)
	{
		float depth;
		if (!SATFacePolygonGlobal(colA, colB, colA->m_faces[i], depth))
			return false;

		if (depth < minDepthA)
		{
			minDepthA = depth;
			faceA = colA->m_faces[i].first;
			sepAxisA = colA->m_faces[i].second;
		}
	}

	//For each face normal of collider B
	for (size_t i = 0; i < colB->m_faces.size(); ++i)
	{
		float depth;
		if (!SATFacePolygonGlobal(colB, colA, colB->m_faces[i], depth))
			return false;

		if (depth < minDepthB)
		{
			minDepthB = depth;
			faceB = colB->m_faces[i].first;
			sepAxisB = colB->m_faces[i].second;
		}
	}

	float minDepth;
	std::vector<size_t> face;
	glm::vec3 sepAxis;

	if (minDepthA < minDepthB * 1.002f + 0.0005f)
	{
		//use axis in col A
		flip = false;
```

```
        minDepth = std::min(minDepthA,minDepthB);
        face = faceA;
        sepAxis = sepAxisA;
    }
    else
    {
        flip = true;
        minDepth = std::min(minDepthA, minDepthB);
        face = faceB;
        sepAxis = sepAxisB;
    }

    return CreateFaceContact(sepAxis, flip, face, colA, colB, cp);
}
```

Creating Contact Points (Helpers.h)

```
static void ClipPolygonWithPlane(const std::vector<glm::vec3>& polygonVerts,
                                 const glm::vec3& planePoint,
                                 const glm::vec3& planeNormal,
                                 std::vector<glm::vec3>& out)
{
    size_t start = polygonVerts.size() - 1;
    float dot_normpoint = glm::dot(planeNormal, planePoint);

    float dot_start = glm::dot((polygonVerts[start] - planePoint)
                                , planeNormal);

    for (size_t end = 0; end < polygonVerts.size(); ++end)
    {
        glm::vec3 v0 = polygonVerts[start];
        glm::vec3 v1 = polygonVerts[end];

        float dot_end = glm::dot((v1 - planePoint), planeNormal);
        if (dot_end >= 0.f)
        {
            if (dot_start < 0.f)
            {
                float t = PlaneLineIntersection(v0, v1,
                                                dot_normpoint,
                                                planeNormal);

                if (t >= 0.f && t <= 1.f)
                    out.push_back(v0 + t * (v1 - v0));
                else
                    out.push_back(v1);
            }

            out.push_back(v1);
        }
        else
        {
            if (dot_start >= 0.f)
```

```
                {
                     float t = PlaneLineIntersection(v0, v1,
                                                     -dot_normpoint,
                                                     -planeNormal);
                     if (t >= 0.f && t <= 1.f)
                          out.push_back(v0 + t * (v1 - v0));
                     else
                          out.push_back(v0);
                }
           }

           start = end;
           dot_start = dot_end;
      }
}

static bool CreateFaceContact(const glm::vec3& sepNormal, bool flip,
                              const std::vector<size_t>& face,
                              std::shared_ptr<Collider> colA,
                              std::shared_ptr<Collider> colB,
                              std::vector<ContactPoint>& cp)
{
      auto reference = flip ? colB : colA;
      auto incident = flip ? colA : colB;

      //Find incident faces
      std::vector<std::pair<std::vector<size_t>, glm::vec3>> incidentFaces;
      if (incident->m_type == BoundingType::CONVEX)
           incidentFaces =

std::static_pointer_cast<ConvexCollider>(incident)->m_faces;

      std::vector<glm::vec3> incidentVertices;
      if (incident->m_type == BoundingType::CONVEX)
           incidentVertices =

std::static_pointer_cast<ConvexCollider>(incident)->m_vertices;

      std::vector<glm::vec3> referenceVertices;
      if (reference->m_type == BoundingType::CONVEX)
           referenceVertices =

std::static_pointer_cast<ConvexCollider>(reference)->m_vertices;

      size_t incidentFaceIndex = FindMostAntiParallelFace(incident,
                                                          sepNormal);
      auto incidentFace = incidentFaces[incidentFaceIndex];

      glm::vec3 normalWorld = sepNormal;
      std::vector<glm::vec3> verticesTemp1;
      std::vector<glm::vec3> verticesTemp2;
      for (size_t i = 0; i < incidentFace.first.size(); ++i)
      {
           glm::vec3 faceVertIncident =
```

```
incidentVertices[incidentFace.first[i]];
			verticesTemp1.push_back(faceVertIncident);
		}

		size_t curr_index = 0;
		size_t number = 0;
		glm::vec3 edgeV1 = referenceVertices[face[curr_index]];
		bool vertice1Input = false;

		//Clipping
		do {

			vertice1Input = !vertice1Input;

			++curr_index;
			if (curr_index == face.size())
				curr_index = 0;

			glm::vec3 edgeV2 = referenceVertices[face[curr_index]];
			glm::vec3 edgeDirection = glm::normalize(edgeV2 - edgeV1);

			glm::vec3 planeNormal = glm::cross(sepNormal, edgeDirection);

			if (vertice1Input)
				ClipPolygonWithPlane(verticesTemp1, edgeV1,
							planeNormal, verticesTemp2);
			else
				ClipPolygonWithPlane(verticesTemp2, edgeV1,
							planeNormal, verticesTemp1);

			edgeV1 = edgeV2;

			if (vertice1Input)
			{
				verticesTemp1.clear();
				number = verticesTemp2.size();
			}
			else
			{
				verticesTemp2.clear();
				number = verticesTemp1.size();
			}

		} while (curr_index != 0 && number > 0);

		std::vector<glm::vec3>& clippedPoints = vertice1Input ?
							verticesTemp2 : verticesTemp1;

		glm::vec3 referenceFaceVert =
					glm::vec3(glm::vec4(referenceVertices[face[0]],
1.f));
		bool found = false;
		for (size_t i = 0; i < clippedPoints.size(); ++i)
		{
			glm::vec3 clippedInWorld = clippedPoints[i];
```

```
            float penetration = glm::dot((referenceFaceVert -
clippedInWorld),
                                                sepNormal);

            if (penetration > 0.f)
            {
                  found = true;

                  glm::vec3 contactPointIncident = clippedPoints[i];
                  glm::vec3 contactPointReference =

ProjectPointToPlane(clippedPoints[i],
                                                                  sepNormal,

referenceFaceVert);
                  ContactPoint c;
                  c.contactNormal = normalWorld;
                  c.contactPointA = contactPointIncident;
                  c.contactPointB = contactPointReference;
                  c.penetrationDepth = penetration;
                  cp.push_back(c);
            }
      }

      return found;
}
```

Collision Resolution

For collision resolution, I used the Sequential Impulses Method. Briefly, the collision resolution process of the Sequential Impulses Method is split into a series of smaller sub-steps, where the impulses are applied sequentially to each contact point. Thus, the contact constraints are solved for each sub-step using an iterative solver, which adjusts the impulses until the contact forces are balanced and the objects have separated or come to rest. As advantages of the Sequential Impulses, it supports stacking, friction, and constraints with multiple contact points while maintaining stability and accuracy.

There are impulses that can be accumulated, normal impulse and tangent impulse. Tangent impulse is for the friction so only normal impulse is implemented in this simulation. Normal impulse in each contact point is calculated using bias impulse, that allows the slop, relative velocity, normal mass, and contact normal. The equation is below :

$$P_n = \max\left(\frac{-\Delta\bar{\mathbf{v}} \cdot \mathbf{n} + v_{bias}}{k_n}, 0\right)$$

, where

$$v_{bias} = \frac{\beta}{\Delta t}\max\left(0, \delta - \delta_{slop}\right)$$

$$\Delta\overline{\mathbf{v}} = \overline{\mathbf{v}}_2 + \overline{\boldsymbol{\omega}}_2 \times \mathbf{r}_2 - \overline{\mathbf{v}}_1 - \overline{\boldsymbol{\omega}}_1 \times \mathbf{r}_1$$

$$k_n = \frac{1}{m_1} + \frac{1}{m_2} + \left[I_1^{-1}\left(\mathbf{r}_1 \times \mathbf{n}\right) \times \mathbf{r}_1 + I_2^{-1}\left(\mathbf{r}_2 \times \mathbf{n}\right) \times \mathbf{r}_2\right] \cdot \mathbf{n}$$

The collision resolution starts with a warm start before actually resolving the collisions. There is also the contact position solver that is performed after position integration. It will not be seen on the following code snippet since it's still in the process.

Initialize Constraints (Physics.cpp)

```
void Physics::InitializeConstraints()
{
  for (size_t i = 0; i < m_CollisionQueue.size(); ++i)
  {
     auto curr = m_CollisionQueue[i];
     auto colA = curr->a->m_collider;
     auto colB = curr->b->m_collider;

     glm::vec3& vA = curr->a->Velocity();
     glm::vec3& vB = curr->b->Velocity();
     glm::vec3& wA = curr->a->AngularVelocity();
     glm::vec3& wB = curr->b->AngularVelocity();
     glm::mat4 iIA = curr->a->GetInverseIntertiaTensor();
     glm::mat4 iIB = curr->b->GetInverseIntertiaTensor();

     float iMA = curr->a->GetInverseMass();
     float iMB = curr->b->GetInverseMass();

     for (size_t j = 0; j < curr->contactPoints.size(); ++j)
     {
        auto& contact = curr->contactPoints[j];
        glm::vec3 rA = contact.contactPointA - colA->m_position;
        glm::vec3 rB = contact.contactPointB - colB->m_position;

        // Kn = 1/m1 + 1/m2 + [I1^-1 (r1 x n) x r + I2*-1 (r2 x n) x r2] * n
        glm::vec3 K = glm::vec3(iIA * glm::vec4(glm::cross(glm::cross(rA,
                                        contact.contactNormal), rA), 1.f)
                        + iIB * glm::vec4(glm::cross(glm::cross(rB,
                                        contact.contactNormal), rB), 1.f));

        float Kn = iMA + iMB + glm::dot(K, contact.contactNormal);
        contact.normalMass = (Kn > 0.f) ? (1.0f / Kn) : 0.f;

        contact.velocityBias = 0.f;
        //relative velocity = v2 + w2 x r2 - v1 - w1 x r1
        float rVel = glm::dot(contact.contactNormal, vB
                        + glm::cross(wB, rB) - vA - glm::cross(wA, rA));
        if (rVel < -0.5f)
           contact.velocityBias = -contact.restitution * rVel;
```

```
            contact.posA = curr->a->m_collider->m_position;
            contact.posB = curr->b->m_collider->m_position;
      }
   }
}
```

Warm Start (Physics.cpp)

```
void Physics::WarmStart()
{
      for (size_t i = 0; i < m_CollisionQueue.size(); ++i)
      {
            auto curr = m_CollisionQueue[i];

            glm::vec3& vA = curr->a->Velocity();
            glm::vec3& vB = curr->b->Velocity();
            glm::vec3& wA = curr->a->AngularVelocity();
            glm::vec3& wB = curr->b->AngularVelocity();

            for (size_t j = 0; j < curr->contactPoints.size(); ++j)
            {
                  auto& contact = curr->contactPoints[j];
                  glm::vec3 rA = contact.contactPointA
                               - curr->a->m_collider->m_position;
                  glm::vec3 rB = contact.contactPointB
                               - curr->b->m_collider->m_position;

                  glm::vec3 P = contact.normalImpulse * contact.contactNormal;
                  wA -= glm::vec3(iIA * glm::vec4(glm::cross(rA, P), 1.f));
                  vA -= iMA * P;
                  wB += glm::vec3(iIB * glm::vec4(glm::cross(rB, P), 1.f));
                  vB += iMB * P;
            }
      }
}
```

Resolve (Physics.cpp)

```
void Physics::Resolve(std::shared_ptr<CollisionData> col, float dt)
{
      auto colA = col->a->m_collider;
      auto colB = col->b->m_collider;

      float iMA = col->a->GetInverseMass();
      float iMB = col->b->GetInverseMass();

      glm::mat4 iIA = col->a->GetInverseIntertiaTensor();
      glm::mat4 iIB = col->b->GetInverseIntertiaTensor();

      glm::vec3 vA = col->a->Velocity();
      glm::vec3 vB = col->b->Velocity();
      glm::vec3 wA = col->a->AngularVelocity();
      glm::vec3 wB = col->b->AngularVelocity();
```

```
	for (size_t i = 0; i < col->contactPoints.size(); ++i)
	{
		auto& contact = col->contactPoints[i];

		glm::vec3 rA = contact.contactPointA - colA->m_position;
		glm::vec3 rB = contact.contactPointB - colB->m_position;

		glm::vec3 vDelta = vB + glm::cross(wB, rB)
		                      - vA - glm::cross(wA, rA);
		float dotDN = glm::dot(vDelta, contact.contactNormal);

		float biasImpulse = 0.f;
		//slop = 0.01, bias factor = 0.3
		if (contact.penetrationDepth > 0.01)
		    biasImpulse = -(0.3f / dt)
		                  * std::max(0.f, contact.penetrationDepth -
0.01f);
		float b = biasImpulse + contact.velocityBias;

		//current delta impulse
		float deltaLambda = -(dotDN + b) * contact.normalMass;
		float tempLambda = contact.normalImpulse;
		contact.normalImpulse = std::max(contact.normalImpulse
		                                 + deltaLambda,
0.f);
		deltaLambda = contact.normalImpulse - tempLambda;

		glm::vec3 P = deltaLambda * contact.contactNormal;
		vA -= iMA * P;
		wA -= glm::vec3(iIA * glm::vec4(glm::cross(rA, P), 1.f));
		vB += iMB * P;
		wB += glm::vec3(iIB * glm::vec4(glm::cross(rB, P), 1.f));
	}

	col->a->Velocity() = vA;
	col->a->AngularVelocity() = wA;
	col->b->Velocity() = vB;
	col->b->AngularVelocity() = wB;
}
```

Collision Resolution (Physics.cpp)

```
void Physics::SolveCollisions(float dt)
{
	if(!m_CollisionQueue.empty())
	{
		InitializeConstraints();
		WarmStart();

		for (int j = 0; j < m_velocitySolveIt; ++j)
		{
			for (int i = 0; i < m_CollisionQueue.size(); ++i)
			{
```

```
                    auto col = m_CollisionQueue[i];
                    if (col->collided)
                         Resolve(col, dt);
               }
          }

          //Integrate the position
          for (int j = 0; j < m_CollisionQueue.size(); ++j)
          {
               auto col = m_CollisionQueue[j];
               glm::quat newRotA, newRotB;
               if (col->a->IsDynamic())
               {
                    glm::quat newRotA = glm::quat(0.5f * dt
                                        * col->a->AngularVelocity());
                    col->a->m_collider->m_rotation *= newRotA;
               }
               if (col->b->IsDynamic())
               {
                    glm::quat newRotB = glm::quat(0.5f * dt
                                          * col->b->AngularVelocity());
                    col->b->m_collider->m_rotation *= newRotB;
               }

               col->a->m_collider->m_position += dt * col->a->Velocity();
               col->b->m_collider->m_position += dt * col->a->Velocity();
               col->a->UpdateInverseInertiaTensor();
               col->b->UpdateInverseInertiaTensor();
          }
          //Solve positions
          for (size_t j = 0; j < m_positionSolveIt; ++j)
          {
               bool solved = true;
               for (size_t i = 0; i < m_CollisionQueue.size(); ++i)
               {
                    auto col = m_CollisionQueue[i];
                     solved = SolvePositionConstraints(col, dt);
               }
               //If all position is solved, stop
               if (solved)
                    break;
          }
          m_CollisionQueue.clear();
     }
}
```

Other Codes in Physics.cpp

Physics class (Physics.h)

```
class Physics
{
```

```
public:
...
	AABBDynamicTree* tree;

protected:
	std::vector<Object*> m_DynamicPhysicsObjects;
	std::vector<Object*> m_StaticPhysicsObjects;
	std::queue<CollisionData> m_CollisionQueue;
private:
...

};

extern std::unique_ptr<Physics> g_Physics;
```

Integrate function (Physics.cpp)

```
void Physics::Integrate(float dt)
{
	for (auto obj : m_DynamicPhysicsObjects)
	{
		RigidBody& rb = obj->rigidbody;
		rb.m_collider->m_color = glm::vec3(0, 0, 1);

		if (!rb.IsDynamic())
			continue;
		rb.SetGravityForce(m_Gravity);

		if (m_EnableGravity && rb.TakesGravity())
			rb.ApplyGravity();

		//Update velocities
		rb.Velocity() += dt * rb.NetForce() * rb.GetInverseMass();
		rb.AngularVelocity() += dt * rb.NetTorque();

		//Update position and rotation
		rb.m_collider->m_position += dt * rb.Velocity();
		rb.m_collider->m_rotation *= glm::quat(0.5f * dt
		                             * rb.AngularVelocity());

		//For debug drawing(velocity)
		rb.BackupForceVector();

		//Clear force
		rb.SetNetForce({ 0.0f,0.0f,0.0f });
		rb.SetNetTorque({ 0.0f,0.0f,0.0f });

		glm::mat4 curr_rot = glm::toMat4(rb.m_collider->m_rotation);
		rb.m_collider->m_objTr = Translate(rb.m_collider->m_position.x,
		                                   rb.m_collider->m_position.y,
		                                   rb.m_collider->m_position.z)
		                         * curr_rot
		                         * Scale(rb.m_collider->m_scale.x,
```

```
                                                    rb.m_collider->m_scale.y,
                                                    rb.m_collider->m_scale.z);
            //Update colliders
            rb.m_collider->Update();
            rb.m_collider->UpdateAABB();
      }

      for (auto obj : m_StaticPhysicsObjects)
      {
            RigidBody& rb = obj->rigidbody;
            rb.m_collider->Update();
            rb.m_collider->UpdateAABB();
      }

      //Update tree
      tree->Update();
}
```

Update function (Physics.cpp)

```
void Physics::Update(float dt)
{
      // Do dynamic updates
      Integrate(dt);

      // Then solve for collisions
      DetectCollisions(dt);
      SolveCollisions(dt);
}
```

Debug Draw

The physics engine includes various debugging tools to help with testing. The engine provides play, pause, and step buttons to control the simulation's speed and progress. Additionally, the engine offers the ability to choose the drawing of colliders, the dynamic AABB tree, and velocities to visualize how objects are interacting with the environment.

For collision detection, there are two debug settings: one for the broad phase and one for the narrow phase. If a collision is detected during broad phase collision detection, the corresponding AABB tree node will be colored red in the drawing tree mode. Similarly, during narrow phase collision detection, the colliders' colors will be set to red in the drawing collider mode if the collision is detected. Also, the generated contact points are rendered as points if the 'Draw Contact Points' is checked.

The engine also includes an 'Object' section in UI that enables users to translate, rotate, and scale the objects in the scene, as well as add or remove objects during runtime. In Stress Test mode, new objects will be spawned in every frame until the FPS goes to below the 30.

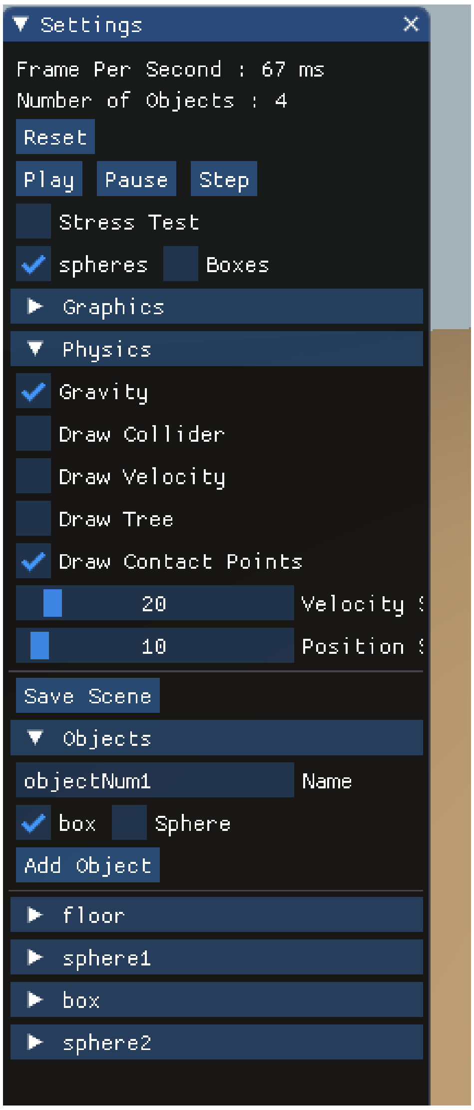


Figure 3. The UI for debugging the engine.

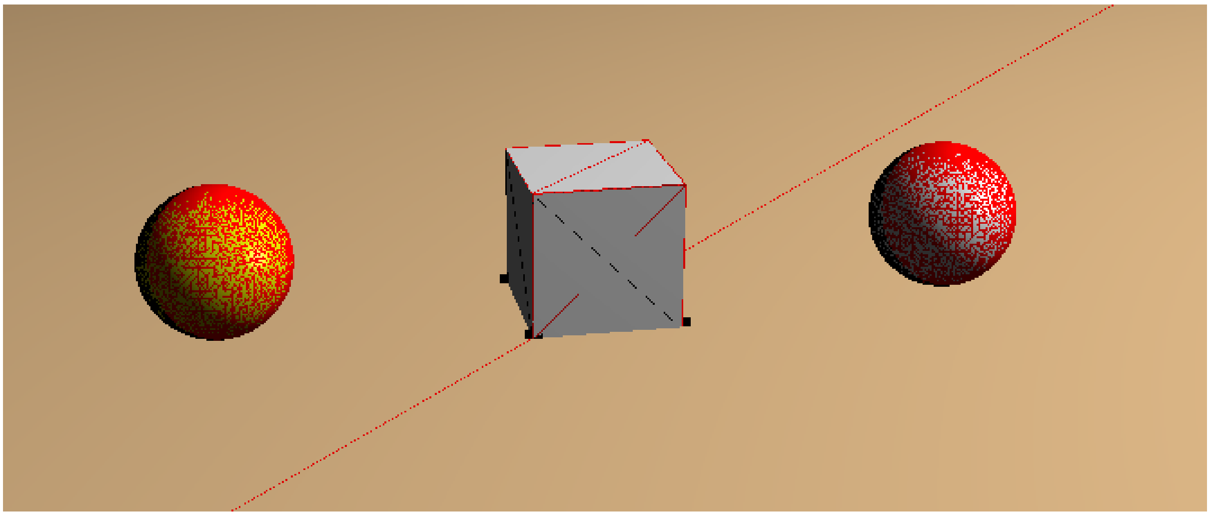


Figure 4. Debug drawing for narrow phase collision detection.

Conclusion

My team's game engine currently lacks broad phase collision detection. However, through this project, I have gained knowledge on how to optimize collision detection using this technique. Although this simulation currently only uses spheres and cubes, my implementation of the Separating Axis Theorem(SAT) should work for other polygon shapes as well. Thus, testing for other convex shapes is a potential future task. Additionally, there is currently no friction implemented in this simulation, so that could also be a future task to tackle.

References

Catto, E. (n.d.). *Box2D | A 2D physics engine for games*. From https://box2d.org/files/ErinCatto_SequentialImpulses_GDC2006.pdf

Gregorius, D. (n.d.). *Steam*. From http://media.steampowered.com/apps/valve/2015/DirkGregorius_Contacts.pdf

Catto, E. (n.d.). *Box2D | A 2D physics engine for games*. From https://box2d.org/files/ErinCatto_DynamicBVH_GDC2019.pdf

Gregorius, D. (n.d.). *Steam*. From http://media.steampowered.com/apps/valve/2015/DirkGregorius_Contacts.pdf

Catto, E. (n.d.). *Box2D | A 2D physics engine for games*. From https://box2d.org/files/ErinCatto_IterativeDynamics_GDC2005.pdf

Intersection of convex objects: The method of separating axes. (n.d.). From https://geometrictools.com/Documentation/MethodOfSeparatingAxes.pdf

Catto, E. (n.d.). *Box2D | A 2D physics engine for games*. From https://box2d.org/files/ErinCatto_ContactManifolds_GDC2007.pdf