MathsDemo Reference Document

By Leonard Andrew Spencer, 2017

# Sections

**Main Loop** 3

**Scene Hierarchy and Collision** 3

**Tank Game** 9

**Robot Arm Game** 10

**3D Game** 11

# Main Loop

The main method and Application2D class

## Classes

**Application2D**  4

## Functions

### Main

int main()

Entry point of the program. Creates and runs an Application2D object.

# Application2D

class Application2D : public aie::Application

A Bootstrap application running the games in the demo

## Functions

### Startup

virtual bool startup()

Creates font, renderer, and each of the games, then sets current game as m\_tankGame and calls its startup function.

### Shutdown

virtual void shutdown()

Destroys font, renderer, and games.

### Update

virtual void update(float deltaTime)

This function will be repeatedly called until the application is quit.

Checks user input for commands to quit the application, switch game modes, or reset the program. If switching modes, m\_game is set to that game and its startup function is called. On a reset, all games are recreated and the game is set back to m\_tankGame.

Then, the current game’s update function is called.

### Draw

virtual void draw()

This function will be repeatedly called until the application is quit.

The function clears the screen, calls the current game’s draw function, then prints the current frames per second to the top left of the screen.

## Members

**m\_2dRenderer:** A Renderer2D for drawing sprites to the screen

**m\_font:** Font used for writing framerate information

**m\_game:** GameMode currently being played

**m\_tankGame:** TankGame

**m\_armGame:** RobotArmGame

**m\_3dGame:** Game3D

**m\_cameraX, m\_cameraY:** Camera position

# Scene Hierarchy and Collision

Classes for collision detection and the scene graph hierarchy.

## Classes

**Plane 6**

**Collider 7**

**AABox 8**

**CircleCollider 8**

**OBox 9**

**Ray 11**

**SceneObject 10**

**SceneObject3D 11**

# Plane

class Plane

A line bisecting a 2d space. In this application it is used for testing whether a collider is out of bounds.

## Member Variables

**m\_normal** Vector2 normal to the plane, defining its orientation and which side is the front of the plane

**m\_offset** Distance from the plane to the origin. If negative, the origin is behind the plane

## Member Functions

DistanceToPlanefloat distanceToPlane(Vector2 point)

Returns the distance between a point and the plane, with a positive value meaning a point in front of the plane and a negative value meaning a point behind the plane.

# Collider

class Collider

An abstract class for colliders. By using virtual functions, collisions can be checked with a collider of unknown type. Aside from ray collisions, the functions return a pair containing whether a collision occurred and a penetration vector. The penetration vector’s direction is from the collider’s edge to the edge of whatever it collided with, such that translating the collider by the penetration vector would cause them to no longer be colliding.

## Member Functions

### DoesCollide

virtual std::pair<bool, Vector2> doesCollide(Collider\* other) = 0  
virtual std::pair<bool, Vector2> doesCollide(Vector2 point) = 0  
virtual std::pair<bool, Vector2> doesCollide(Plane plane) = 0

virtual std::pair<bool, Vector2> doesCollideWithAABox(AABox\* box) = 0

virtual std::pair<bool, Vector2> doesCollideWithOBox(OBox\* box) = 0

virtual std::pair<bool, Vector2> doesCollideWithCircle(CircleCollider\* circle) = 0

Returns whether the two are colliding, and if so a penetration vector such that translating the collider by that vector would cause it to no longer be colliding with the object passed to it.

### IsHitByRay

virtual bool isHitByRay(Ray\* ray) = 0

Returns whether the ray hits the collider

## Derived Classes

**AABox 7**

**CircleCollider 8**

**OBox 9**

**SceneObject 10**

# AABox

class AABox : public Collider

An axis aligned bounding box.

## Member **Variables**

**m\_min** Corner of box with minimum x and y coordinates

**m\_max** Corner of box with maximum x and y coordinates

## Member Functions

### DoesCollide(Collider)

virtual std::pair<bool, Vector2> doesCollide(Collider\* other)

Calls the other collider’s doesCollideWithAABox method, and reverses the direction of the penetration vector

### DoesCollide(Point)

virtual std::pair<bool, Vector2> doesCollide(Vector2 point)

Tests whether the point is within the AABox. If so, the penetration vector is the smallest distance between a side of the box and the point.

### DoesCollide(Plane)

virtual std::pair<bool, Vector2> doesCollide(Plane plane)

Gets the distance from the plane for each corner of the box, and selects the lowest. If positive, no part of the box collides with the plane. If negative, then it is the furthest point behind the plane, and so the penetration vector is equal to the plane’s normal scaled by absolute value of that distance.

### DoesCollideWithAABox

virtual std::pair<bool, Vector2> doesCollideWithAABox(AABox\* box)

Tests whether the minimum corner of one box is greater than the maximum corner of the other in any dimension, and if so they do not collide. Otherwise, the penetration vector is the smallest distance along an axis between a maximum and minimum edge of the boxes.

### Does CollideWithOBox

virtual std::pair<bool, Vector2> doesCollideWithOBox(OBox\* box)

This calls the OBox’s doesCollideWithAABox method

### DoesCollideWithCircle

virtual std::pair<bool, Vector2> doesCollideWithCircle(CircleCollider\* circle)

This calls the CircleCollider’s doesCollideWithAABox method

### GetCorners

std::tuple<Vector2, Vector2, Vector2, Vector2> getCorners()

Returns a tuple of all four of the box’s corners (not just maximum and minimum) such that each corner is adjacent to the ones before and after it, and corners 0 and 3 are adjacent. In this implementation corner 0 is m\_min and corner 3 is m\_max.

# CircleCollider

class CircleCollider : public Collider

A circle

## Member Variables

**m\_centre** Centre of circle

**m\_radius** Radius of circle

## Member Functions

### DoesCollide(Collider)

virtual std::pair<bool, Vector2> doesCollide(Collider\* other)

Calls the other collider’s doesCollideWithCircleCollider method, and reverses the direction of the penetration vector

### DoesCollide(Point)

virtual std::pair<bool, Vector2> doesCollide(Vector2 point)

Tests whether the distance between the point and circle’s centre is less than its radius, and if so the penetration vector is the difference between the point and the nearest point to it on the edge of the circle.

### DoesCollide(Plane)

virtual std::pair<bool, Vector2> doesCollide(Plane plane)

Gets the distance between the circle’s centre and the plane, the subtracts the circle’s radius. It this value is negative, the circle and plane collide. The penetration vector is equal to the plane’s normal scaled by this distance times -1.

### DoesCollideWithAABox

virtual std::pair<bool, Vector2> doesCollideWithAABox(AABox\* box)

First tests whether the circle’s centre is inside the box. If so, the penetration vector is equal to the penetration of the centre in the box, plus the circle’s radius.

Otherwise, the closest point on the box to the circle’s centre is found by clamping the value of the circle’s centre between the box’s maximum and minimum corners. This point is tested against the circle.

The test is done in this way so that when the centre of the circle is within the box, the penetration vector will be correctly calculated. If the closest point on the box were used in this case, the penetration vector would always have a magnitude equal to the circle’s radius.

### Does CollideWithOBox

virtual std::pair<bool, Vector2> doesCollideWithOBox(OBox\* box)

This calls the OBox’s doesCollideWithCircle method

### DoesCollideWithCircle

virtual std::pair<bool, Vector2> doesCollideWithCircle(CircleCollider\* circle)

First the vector between the two circle’s centres is calculated. If the magnitude of this vector is less than the sum of their radii, the circles collide. Then, the points on that vector at the edge of each circle are found, and the difference between them is the penetration vector.

# OBox

class OBox : public Collider

Oriented bounding box.

## Member Variables

**m\_centre** Centre of the box

**m\_xExtent, m\_yExtent** Half extents of the box

## Member Functions

### DoesCollide(Collider)

virtual std::pair<bool, Vector2> doesCollide(Collider\* other)

Calls the other collider’s doesCollideWithOBox method, and reverses the direction of the penetration vector

### DoesCollide(Point)

virtual std::pair<bool, Vector2> doesCollide(Vector2 point)

Calculates the projection of the difference between the point and box’s centre along each half extent. If this is greater than the magnitude of that extend, the point lies outside the box. Otherwise, the vector from the end of an extent to the point projected onto that extent is found, and the smallest is passed as the penetration vector. If this is zero, the point lies exactly on an edge, and so a collision is not considered to have occurred.

### DoesCollide(Plane)

virtual std::pair<bool, Vector2> doesCollide(Plane plane)

Gets the distance from the plane for each corner of the box, and selects the lowest. If positive, no part of the box collides with the plane. If negative, then it is the furthest point behind the plane, and so the penetration vector is equal to the plane’s normal scaled by absolute value of that distance.

### DoesCollideWithAABox

virtual std::pair<bool, Vector2> doesCollideWithAABox(AABox\* box)

Tests collision between this and an AABox using the separating axis theorem. The corners of both boxes are used to calculate the axes to test. For each axis, the boxes have each corner projected onto the axis and the maximum and minimum projections are compared to determine if they overlap. If any axis does not have an overlap, the boxes do not collide, otherwise the penetration vector is along the axis with lowest overlap, with magnitude equal to that overlap.

### Does CollideWithOBox

virtual std::pair<bool, Vector2> doesCollideWithOBox(OBox\* box)

Tests collision between this and another OBox using the separating axis theorem. The corners of both boxes are used to calculate the axes to test. For each axis, the boxes have each corner projected onto the axis and the maximum and minimum projections are compared to determine if they overlap. If any axis does not have an overlap, the boxes do not collide, otherwise the penetration vector is along the axis with lowest overlap, with magnitude equal to that overlap.

### DoesCollideWithCircle

virtual std::pair<bool, Vector2> doesCollideWithCircle(CircleCollider\* circle)

Tests collision between this and a CircleCollider using the separating axis theorem. The corners of this box are used to calculate the axes to test. For each axis, the box’s corners are projected onto the axis, and the maximum and minimum projections are compared to the circle’s centre projected onto the axis, plus or minus the circle’s radius. If there is no overlap between these values, the box and circle do not collide. Otherwise, the penetration vector is along the axis with lowest overlap with magnitude equal to that overlap.

### GetCorners

std::tuple<Vector2, Vector2, Vector2, Vector2> getCorners()

Returns a tuple of all four of the box’s corners (not just maximum and minimum) such that each corner is adjacent to the ones before and after it, and corners 0 and 3 are adjacent.

### GetBoxMatrix

Matrix3 getBoxMatrix()

Returns the half-extents and centre of the OBox as a transformation matrix.

### GetInverseTransform

### Matrix3 getInverseTransform()

Returns the inverse transformation matrix of a matrix defined by the OBox’s half-extents and centre. Applying the result of this function to the OBox would transform it into an axis aligned box at the origin with corners [-1,-1] and [1,1]. This is used in testing ray collisions with the OBox.

# Ray

class Ray

A ray extending from a point

## Member Variables

**m\_origin** Origin of the ray

**m\_direction** Direction in which the ray extends as a unit vector

## Member Functions

### FindClosestPoint

### Vector2 findClosestPoint(Vector2 Point)

Projects the difference between the ray’s origin and the point onto the ray’s direction. If this projection is negative, the nearest point is the ray’s origin, otherwise it is the point along the ray that distance from the origin.

### DoesCollide(Collider)

### bool doesCollide(Collider\* collider)

Calls the isHitByRay method of the collider, which will call the appropriate doesCollide method of this ray.

### DoesCollide(CircleCollider)

### bool doesCollide(CircleCollider\* circle)

Finds the closest point on the ray to the circle’s centre. If the distance between that point and the centre is less than the circle’s radius, returns true.

### DoesCollide(AABox)

bool Ray::doesCollide(AABox\* box)

Tests whether a ray collides with an axis aligned box.

The distances from the origin at which a line drawn through the origin along the ray’s direction at which it shares an X or Y coordinate with a side of the box are calculated. The lowest distance at which both an X and Y coordinate have been equalled is called the entry point, and set to 0 if less than 0. The lowest distance at which both the minimum and maximum values of one coordinate have been passed is called the exit point. If the exit point is greater than the entry point, the ray intersects with the box. Otherwise, the ray must have passed by the box in one axis before reaching it in the other axis, and so it does not intersect.

### DoesCollide(OBox)

### bool doesCollide(OBox\* box)

Tests whether a ray collides with an oriented box.

The getInverseTransform method of the box is called, and used to transform the direction and origin of the ray. This transformed ray is tested against an axis aligned box with corners [-1,-1] and [1,1]. If the transformed ray collides with that box, then this ray collides with the oriented box.

# SceneObject

class SceneObject

A 2D object within a hierarchical scene graph.

## Member Variables

**m\_globalTransform:** Transformation matrix relative to global coordinate system. Recalculated every update.

**m\_localTransform:** Transformation matrix relative to parent

**m\_parent:** Pointer to the parent of this SceneObject

**m\_children:** Vector of pointers to the children of this SceneObject

**m\_toAdd:** Vector of pointers to SceneObjects which will be added as children at the next end of frame

**m\_toDelete:** Vector of pointers to children which will be removed from m\_children and deleted at the next end of frame

**m\_toTransfer:** Vector of pairs containing a child and a SceneObject to transfer that child to at the next end of frame

**m\_childrenLocked:** Flag indicating that m\_children is being iterated over, and so cannot be modified

**m\_deletionFlag:** Flag indicating that this SceneObject is to be deleted at the next end of frame

**m\_transferFlag:** Flag indicating that this SceneObject is to be reparented at the next end of frame.

**m\_transferTargetFlag:** Flag indicating that another SceneObject is to be reparented to this at the next end of frame

**m\_collider:** Collider used to detect collisions with this SceneObject. Recalculated every update

## Member Functions

### addChild

bool addChild(SceneObject\* child)

Tries to add a SceneObject to m\_children. If m\_children is not locked, the child is pushed onto it, otherwise it is pushed onto m\_toAdd so it can be added at the next end of frame.

If child already has a parent, or a SceneObject is being added to itself, this returns false instead.

### removeChild

bool removeChild(SceneObject\* child)

Tries to remove a child from m\_children and delete it. First, the child and all its descendants are checked for m\_transferFlag and m\_transferTargetFlag. If either flag is true, it is not safe to delete the child and the function returns false. Otherwise, the child is removed from m\_children and deleted (if m\_children is not locked), or added to m\_toDelete and has its deletionFlag set to true;

### TransferChild

bool transferChild(SceneObject\* child, SceneObject\* target)

Tells the SceneObject to reparent child to target at the next end of frame. Both are checked to ensure they aren’t flagged for deletion and the child isn’t flagged to be transferred already. If not, child is flagged with m\_transferFlag and target is flagged with m\_transferTargetFlag, and a pair of child and target is pushed onto m\_toTransfer.

### Rotate, Scale, Translate, Transform

void rotate(float angle)  
void scale(const Vector2& proportions)  
void translate(const Vector2& vec)  
void transform(const Matrix3& transformation)

Perform the indicated transformation on SceneObject. Note that these all apply the transformation after the current localTransform.

### GlobalTranslate

void globalTranslate(const Vector2& vec)

Changes localTransform such that the SceneObject’s globalTransform will be translated by vec. This is done by transforming vec by the inverse of the SceneObject’s parent’s globalTransform. The transformed vector is then added to the translation axis of localTransform.

### Update

virtual void update(float deltaTime)

Calls calculateGlobalTransform and setupCollider, then updates all children. If this is the root, it also calls updateChildList to perform any queued changes within the scene graph.

### Draw

virtual void draw(aie::Renderer2D\* renderer)

Draws all children.

### GetDescendants

std::vector<SceneObject\*> getDescendants()

Returns a vector containing all descendants of this SceneObject. Each child is added to the vector, along with all of its descendants.

### GetRoot

SceneObject\* getRoot()

Returns a pointer to the root of the scene graph this SceneObject is part of, by traversing up the hierarchy until a SceneObject with no parent is found.

### NotifyCollision, NotifyOutOfBounds

virtual void notifyCollision(SceneObject\* other, Vector2 penetration)  
virtual void notifyOutOfBounds(Vector2 penetration)

These are methods for derived classes to implement responses to collision or being placed out of bounds.

### SetupCollider

virtual void setupCollider()

This method is for derived classes to define their collider. SceneObject has a default AABox as its collider.

### PerformChildTransfer

void performChildTransfer(SceneObject\* child, SceneObject\* target);

Calculates the child’s new local transform (as that which would give the same global transform when multiplied by the target’s global transform), removes the child’s pointer from m\_children, and adds it to target.

### UpdateChildList

void updateChildList()

Performs all end of frame changes to m\_children. First, updateChildList is called on each child SceneObject. This is so any SceneObject being transferred from a descendant to its ancestor will occur in a single update, as the parent will add new children from the toAdd vector later in this function, and all descendants will have their m\_children unlocked after they perform all changes.

Next, performChildTransfer is called for each pair in m\_toTransfer, and every SceneObject in m\_toAdd and m\_toDelete are added or deleted. Each of these vectors is cleared, and the m\_deletionFlag, m\_transferFlag and m\_transferTargetFlag are reset to false.

# SceneObject3D

class SceneObject3D

A 3D object within a hierarchical scene graph.

## Member Variables

**m\_globalTransform:** Transformation matrix relative to global coordinate system. Recalculated every update.

**m\_localTransform:** Transformation matrix relative to parent

**m\_parent:** Pointer to the parent of this SceneObject

**m\_children:** Vector of pointers to the children of this SceneObject

**m\_toAdd:** Vector of pointers to SceneObjects which will be added as children at the next end of frame

**m\_childrenLocked:** Flag indicating that m\_children is being iterated over, and so cannot be modified

## Member Functions

### addChild

bool addChild(SceneObject\* child)

Tries to add a SceneObject to m\_children. If m\_children is not locked, the child is pushed onto it, otherwise it is pushed onto m\_toAdd so it can be added at the next end of frame.

If child already has a parent, or a SceneObject is being added to itself, this returns false instead.

### Update

virtual void update(float deltaTime)

Calls calculateGlobalTransform and setupCollider, then updates all children. If this is the root, it also calls updateChildList to perform any queued changes within the scene graph.

### Draw

virtual void draw(aie::Renderer2D\* renderer)

Draws all children.

### GetDescendants

std::vector<SceneObject\*> getDescendants()

Returns a vector containing all descendants of this SceneObject. Each child is added to the vector, along with all of its descendants.

### UpdateChildList

void updateChildList()

Performs all end of frame changes to m\_children. First, updateChildList is called on each child SceneObject. Then, every SceneObject in m\_toAdd is added to m\_children, and m\_toAdd is cleared.

# Tank Game

## Classes

**Tank 19**

**Bullet 20**

**Wall 21**

**Obstacle 22**

# Tank

class Tank : public SceneObject

A tank controlled by the player.

## Member Functions

### Update

virtual void update(float deltaTime)

### Draw

virtual void draw(aie::Renderer2D\* renderer)

### FireBullet

void fireBullet()

A bullet is created at the end of the tank’s turret, moving away from the tank. It is added to the root of the scene graph.

### NotifyColision

virtual void notifyCollision(SceneObject\* other, Vector2 penetration)

When the tank collides with a wall, it is translated by penetration. When it collides with an Obstacle, it is translated by half the penetration as the Obstacle will also be pushed back.

# Bullet

# Wall

# Obstacle

# Robot Arm Game

# 3D Game

## Classes

**GLMAdaptor 21**

# GLMAdaptor

A class containing static methods for converting lasmath vectors and matrices into glm vectors and matrices, as well as calculating a view matrix and projection matrix using the lasmath Matrix4 class.

## Functions

### LookAt

static lasmath::Matrix4 lookAt(const lasmath::Vector3& eye, const lasmath::Vector3& centre, const lasmath::Vector3& up)

Calculates a view matrix from the vectors passed. First, the camera transformation matrix is created. This is done by translating to the position specified by eye. Then, the matrix is rotated so that it’s Z-Axis is pointing away from centre (the point being looked at), its X-Axis is perpendicular to the Z-Axis and the orientation given by the up argument, and its Y-Axis is equal to the cross product of Z and X.

This matrix is inverted, and its inverse matrix is returned as the view matrix

### Perspective

static lasmath::Matrix4 perspective(float fovy, float aspect, float near, float far)

Calculates a perspective projection matrix from the field of view, aspect ratio, and near and far clipping planes