SimObjects Code Library for Processing 3

Version 2.2 Feb 2020. Simon Schofield

Introduction

The SimObjects code library is a Processing 3 library of code and classes that will help you build 3-D physics-based simulations.

The purpose of SimObjects, is that you can

- populate your 3D world with 3D objects, these can be
  - Objects made via code (e.g. sphere, mesh, terrain)
  - .OBJ models loaded from file
- move the objects around, scale and rotate them
- get their positions
- get their geometry and (simplified) bounding geometry
- determine collisions between objects
- Cast rays which collide with objects
- pick objects using the mouse in 3D
- A handy flying camera, with which you can also render things in 2D
- A “manager” class to contain and manage all your 3D objects

Without using the SimObjects library, you will struggle to recover the geometry from transformed objects in your 3D scene and will have to write all your own RayPicking and Collision calculations.

Before you begin

As a starting point for using the SimObjects Library, you need to

- be familiar with Processing 3 and finding reference material on the existing Processing language
- have completed the “Introduction to P3D” by Dianiel Shiffman (processing.org/tutorials/p3d/).
- You need to be particularly familiar with the Processing types:-

  - **PVector** - a class used for storing 3D vectors and points, and performing maths operations on them. You should also me mindful of passing vectors into functions/methods that then change them in the function; this will result in the passed-in PVector being changed also. Use the .copy() on PVector inside the function to avoid this.
  - **PShape** – a class used for storing and drawing 3D shapes. These shapes can be made in code, or loaded from file.

Including SimObjects library in your project

As of version 1.3 (March 19), the SimObjects code is composed of 4 source files, SimShapes.pde, SimSurfaceMesh.pde, Sim_Functions.pde and SimCamera.pde

You need to download the most recent SimObjects example from NOW (G4G->Content->LearningResources->Latest Version of SimObjects etc), and copy these four files into your new project. They will then appear as three tabs in the Processing IDE window. You should not need to edit these tabs/files, but looking at the will be instructional.

You are now ready to use the library in your own project.

SimObject Reference

In the SimShapes and SimMesh source files these are a number of 3D object classes that you can instantiate and place in your scene.

Currently these are

SimSphere – a sphere object that is also used for bounding geometry in more complex shapes

SimBox – a box object that is also used for bounding geometry in more complex shapes. It can be both Axis Aligned (AABB) and Oriented (OBB).

SimModel - a 3D object that stores a PShape. The PShape can be made from code or loaded from file. The file format used for 3D models is the .OBJ file format. This can be exported from 3DS and other modelling packages.

SimSurfaceMesh – A 3D mesh suitable to make surfaces such as table tops and mountain ranges

There is also a **SimCamera** class, which is not strictly a 3D object, and has it's own section later.

There is also a **SimObjectManager** class, is a sort of database to contain all your 3D objects, and manage them.

All the 3D shapes are derived from the **SimTransform** class

The **SimTransform** class stores and applies 3D transformations to derived classes.

The basic principle is that all derived classes (SimSphere, SimBox, SimModel and SimSurfaceMesh) store an untransformed copy of the shape's geometry. This is sometimes called the *cardinal* geometry.

Using the methods inherited from the **SimTransform** base class, you can transform the shape's geometry using scale, rotate and translate. Once a transform has been applied, you can

- Draw the object with the correct transformation
- Get the transformed geometry, including simplified "bounding" geometry
- You can easily remove the transform to get the untransformed geometry too

Utility Functions – global functions not associated with a class

*PVector **vec**(float x, float y, float z)*

> Returns a PVector object from the three floats x,y,z
>
> Because so much code relies on making 3D vectors and points, this is a useful shorthand function vec(x,y,z) for making PVectors on the fly.

boolean ***nearZero****(float v)*

Returns true if the value of v is close to zero. Works with both positive and negative numbers. It uses Java static variable EPSILON as the smallest possible number.

PVector ***getCameraPosition****()*

Returns the current camera position

void ***setCameraPosition****(PVector camPos, PVector lookAt)*

Sets the camera at point camPos, looking at lookat

SimRay ***getMouseRay****()*

Returns a SimRay (3D ray class) into the current scene at the mouse position. This is useful for picking or interacting with objects in the scene with the mouse.

SimTransform methods

This class is not instantiated on its own, but provides a base class to other Sim Objects.

void ***setTransformAbs****(PVector translate, float scale, float rotateX, float rotateY, float rotateZ)*

return type: void

Sets the absolute transformation (translation, scale and rotation) of the 3D object, from the original cardinal geometry. To return a shape to its original (cardinal) geometry use

myShape.*setTransformAbs(vec(0,0,0), 1, 0,0,0);*

void ***setTransformRel****(PVector translate, float scale, float rotateX, float rotateY, float rotateZ)*

return type: void

Sets relative transformation (translation, scale and rotation) of the 3D object, relative to the previous set transform. This is useful for applying small incremental changes to the transformation over time, e.g. moving the shape a bit each frame.

void ***resetTransform****()*

Sets the current transform to be the identity transform. I.e. no actual transform.

PVector ***transform***(*PVector v)*

Transforms a PVector by the current set transform. It returns a transformed copy of the vector v, so vector v is left unchanged.

PVector[] ***getTransformedVertices****(PVector[] vertices)*

Returns a transformed copy of the input list of vertices (an array of PVectors).

PVector ***getOrigin****()*

Many (but not all) shapes have an origin (such as a sphere, ray etc). This will return the origin transformed by the current transform.

PVector[] ***getExtents****(PVector[] vertices)*

Gets the extents on the list of vertices given under the current transformation. The extents are return as a list of 4 PVectors. In order these are

1. The minimum x,y, and z point
2. The maximum x,y,and z point
3. The centre point between 1 and 2
4. The given vertex which is furthest from the centre point, point 3

PVectors 1 and 2 can be used to provide a bounding box. PVectors 3 and 4 can be used to provide a bounding sphere. Remember the index of these runs 0..3.

boolean ***isRotated****()*

Returns true if the rotation in x,y, or z is anything other than 0.0. This is useful for determining if derived objects are Axis Aligned or not, particularly SimBox's.

boolean ***collidesWith****(SimTransform otherObj)*

Is a method implemented in all sub classes of SimTransform in order to determine collision with ANY OTHER sim object. As of this version we have Sphere/Sphere, Sphere/AABox and AABox/AABox collision working.

boolean ***calcRayIntersection****(SimRay ray)*

Is a method implemented in all sub classes of SimTransform in order to determine the intersection of this object with the ray. As with all ray intersections, once an intersection has been determined as true or false, if true, then the data about the intersection can be recovered from the ray. (See SimRay section).

void ***setID****(String id),*

String ***getID****()*

All classes derived from SimTransform (i.e. all 3D objects) have an ID used to uniquely name the instance of the object. These methods set and get a string to name the object. This can be then used by the SimObjectManager class to identify collisions between objects and ray-hits

SimSphere methods

(Derives from SimShape)

This class is used to be a sphere object in the scene, and is also used as a bounding sphere for other objects.

Please note, that because of an implementation issue, the drawing style of sphere is set at its moment of instantiation. It will not respond to fill() and stroke() etc once made. This is because the drawn sphere is an instance of type “Shape” and are not settable once created. However, a method updateDrawStyle() will force an existing sphere to be remade, therefore adopting the current drawing style.

Constructors

SimSphere*()*

Creates a unit sphere (Origin (0,0,0), and radius 1.0).

SimSphere*(float rad)*

Creates a sphere (Origin (0,0,0), and radius *rad*).

SimSphere*(PVector centre, float radius)*

Creates a sphere with Origin at *centre*, and radius *radius*.

Public member variable

(i.e. you can get and set these with the dot syntax mySphere.levelOfDetail = 20;)

int ***levelOfDetail***

The level of detail with which the sphere is drawn. The default is 10. Values below 6 look very poor.

Methods

void **updateDrawStyle**()

Call this to force the sphere to adopt the latest drawing style and level of detail. Only do this intermittently as it is quite processor intensive.

PVector ***getCentre****()*

Returns the centre of the sphere under the current transform. (same as getOrigin())

float ***getRadius****()*

Returns the radius of the sphere under the current transform – use this in preference to diretly accessing radius variable, as *getRadius*() returns the radius with any scaling applied through the current transform.

boolean ***isPointInside****(PVector p)*

Checks to see if a point represented by PVector *p* is inside the transformed sphere.

boolean ***intersectsSphere*** *(SimSphere otherSphere)*

Checks to see if this sphere is colliding with another sphere. Returns true if they are colliding. This method works on the transformed spheres.

boolean ***calcRayIntersection****(SimRay sr)*

Calculates is the ray sr has intersected the sphere. Returns true if an intersection has been detected. The intersection point can be recovered from the SimRay object, once an intersection has been found, by using *sr.getIntersectionPoint().*

As a ray will always intersect two sides of the sphere, the function *sr.getIntersectionPoint()* returns the nearest point to the ray origin.

void ***drawMe****()*

Draws the transformed sphere with the level of detail specified.

SimBox methods

(Derives from SimShape)

The SimBox class provides a 6-sided box, that can be used on its own, or as the collision geometry for a more complex 3d shape.

It can play the role of an Axis-Aligned Bounding Box (AABB), or an Oriented Bounding Box (OBB).

Constructors

SimBox()

Returns a cubic box with sides length 2, around the scene origin

SimBox(*PVector pmin, PVector pmax****)***

Returns a box defined by the extents pmin and pmax. Initially axis aligned.

Methods

void ***setExtents(****PVector c1, PVector c2****)***

Sets the extents of an existing SimBox to c1->c2. Resets the transforms to the identity, so creates an Axis Aligned Box to start with.

PVector[] ***getExtents()***

Returns the extents of the bounding box. Works for both AABB and OBB's.

Gets the extents of the SimBox under the current transformation. The extents are return as an array of 4 PVectors. In order these are

1. The minimum x,y, and z point
2. The maximum x,y,and z point
3. The centre point between 1 and 2
4. The given vertex which is furthest from the centre point, point 3

PVector ***getCentre****()*

Returns the centre of the transformed SimBox. Works for both AABB and OBB's.

SimBox ***getTransformedCopy****()*

Returns a transformed SimBox, with transforms applied to the vertices.

boolean ***isPointInside****(PVector p)*

Returns true if the point p is inside the SimBox.

NOTE: Only works with AABB's. This will be fixed.

boolean ***calcRayIntersection****(SimRay sr)*

Calculates is the ray sr has intersected the SimBox. Returns true if an intersection has been detected. The intersection point can be recovered from the SimRay object, once an intersection has been found, by using *sr.getIntersectionPoint().*

As a ray will always intersect two faces of the box, the function returns the nearest point to the ray origin.

boolean ***intersectsSphere****(SimSphere s)*

Returns true if this box intersects with SimSphere s

NOTE: Only works with AABB's

boolean ***intersectsBox****(SimBox b)*

Returns true if this box intersects with SimBox b

NOTE: Only works with AABB's

Void ***drawMe****()*

Draws the transformed SimBox

// probably will never have to use this method

SimFacet ***getFacet****(int face)*

Returns a transformed SimFacet of a specific face of the box. The faces are referenced thus:

0 = top, 1 = front, 2 = left, 3 = right, 4 = back, 5 = bottom

SimModel methods

(Derives from SimShape)

This class is used to contain and draw a more complex 3D shape. It is instantiated from a PShape type (see Processing 3, PShape documentation for 3D shapes) , which can be made either through code or by loading a PShape from a .OBJ file. The original PShape is not (usually) altered throughout the life of the object, so represents the cardinal geometry.

The shape can be transformed around the scene and has methods to get the transformed geometry and bounding geometry.

Constructors

SimModel()

Returns an “empty” shape waiting to be initialised with geometry by using the *setWithPShape() method*.

Public member variable (i.e. you can get and set these with the dot)

None

Methods

void ***setWithPShape****(PShape shapeIn)*

Sets the internal geometry of the shape to PShape shapeIn. Please note that this does **not** make an independent copy of the PShape data, but sets an internal reference to the passed in PShape object. Hence, alterations to the same PShape object elsewhere in code will effect this shape. PShape does not have a .copy() method as yet.

PVector[] ***getRawVertices****()*

Returns a list of vertices of the stored PShape. This is useful only for doing collision on and calculating collision geometry. The returned list should not be used to draw or copy the PShape, as it does not contain any of the model’s internal structure.

void ***setPreferredCollisionShape*** *(either “box” or “sphere” string)*

When using the SimModel in a the SimObjectManager, this determines whether you want the collision geometry to be either the bounding-box or bounding-sphere. Default is sphere.

SimSphere ***getBoundingSphere****()*

Returns the bounding SimSphere sphere of the transformed model

void ***drawMe****()*

Draws the transformed model

void ***drawBoundingSphere****()*

Draws the bounding sphere of the transformed model

SimSurfaceMesh methods

(Derives from SimShape)

This class is used to model, contain and draw a 3D mesh suitable for representing surfaces such as table tops and mountain ranges. It is constructed from a mesh of quads, each quad is made of two triangles.

Constructors

SimSurfaceMesh*(int numInX, int numInZ, float scale)*

This creates a mesh of *numInX* * *numInZ* quads. Each quad is *scale* units in size. The mesh is initialised with one corner at (0,0,0) and the other corner at (*numInX**scale, 0, *numInZ**scale)

So, it is initialised as a flat surface in the XZ plane, at Y = 0.

Methods

void ***setHeightsFromImage****(PImage im, float maxAltitude)*

A way to set the Y component of every vertices through the application of a “height map”. The image should be a grey-tone image in image mode RGB or RGBA, and contain values 0.. 255 to get the full range of heights. The values in the image are mapped to the heights of the vertices, where a value of 0 in the image corresponds to a height of 0 in the mesh, and a value of 255 in the image, corresponds to a height of *maxAltitude* in model dimensions.

void ***setTextureMap****(PImage im)*

A very simple texture mapping for the sim surface. No scaling or offset available yet. Will not work if you apply transforms at the moment.

void ***applyTransform****()*

Permanently applies the transformation (set using setTransformAbs()) to the vertices of the mesh. This may be useful, as, once in the correct place, the mesh may not change in position/scale etc. if it represents a mountain range or table top. Hence, the transform only needs to be calculated once at the start, rather than on every frame.

boolean ***intersectsSphere*** *(SimSphere sphere)*

Returns true if any vertices of the mesh are found to be inside the *sphere.* If the sphere is too small compared to the mesh size, then the sphere may pass through the mesh with no collision.

Future version could improve on this by checking the sphere/mesh-triangle intersection.

boolean ***intersectsBox*** *(SimBox box)*

Returns true if any vertices of the mesh are found to be inside the *box.* If the box is too small compared to the mesh size, then the box may pass through the mesh with no collision.

Future version could improve on this by checking the box/mesh-triangle intersection.

void ***drawMesh****()*

Draws the transformed mesh

boolean ***calcRayIntersection****(SimRay sr)*

Calculates if the SimRay sr intersects with any of the mesh triangles. Returns true if so. Once an intersection has been found, the intersection point can be recovered from the SimRay by using sr.intersectionPoint, as *sr* is passed by reference.

Another variable in the mesh called *rayIntersectionTriangleNum* is set upon finding an intersection, which represent the intersected triangle index within the mesh.

SimRay methods

(Derives from SimShape)

This class is used to contain a 3D ray. This is an infinite line in 3D space defined by a point on the line (the *origin*) and a vector representing the direction of the line. It is useful for various picking and collision processes.

Constructors

SimRay()

Returns a default ray origin (0,0,0) direction (0,0,-1)

SimRay*(PVector orig,PVector dir)*

Returns a SimRay object with orgin at *orig*, and direction *dir*

Public member variables

PVector **origin**;

PVector **direction**;

The origin is the point at the start of the ray.

The direction is a normalised vector representing the direction of the ray

Methods

SimRay **copy**()

Returns a copy of the ray

boolean ***calcIntersection****(SimRayIntersectionShape shape)*

Calculates if a this SimRay intersects with the *shape.* These include SimSphere, SimBox and SimMesh

boolean ***isIntersection****()*

Returns true if the most recent intersection calculation was an intersection

PVector ***getIntersectionPoint****()*

Returns the intersection point from the most recent intersection calculation

void **drawMe**()

Draws a thin line along the ray line for 10,000 units. Useful for debugging.

boolean ***collidesWith****(SimTransform any3dObj)*

Returns true if this ray intersects with the 3d object.

int ***getNumIntersections****()*

Returns the number of intersections made. Can be used to calculated inside/outside rules of 3D points.

PVector ***getIntersectionNormal****()*

Returns the surface normal of the intersected surface. Tis will be the nearest surface if many are intersected.

PVector ***getPointAtDistance****(float dist)*

Returns a point on the ray that is dist units away from the origin.

SimCamera methods

The sim camera allows you to draw 2D heads-up style graphics on top of your 3D scene. This would be otherwise difficult as P3D insists on rendering everything in 3D, even erstwhile 2D rectangles etc.

It also provides you with a handy “flying camera mode” useful for navigating around the scene

Once you have instantiated your camera, you will need to update it in the main “draw()” method like so:-

```
SimCamera mycamera;
void setup(){
        mycamera = new SimCamera();
        }
void draw(){
        ….
        mycamera.update();
        ….
        }
```

void ***setViewParameters(float fov, float nearClip, float farClip)***

Sets the viewing parameters of the camera.

Fov - Field of View – the vertical field of view angle in radians

nearClip – the distance from the camera to the near clipping plane

farClip – the distance from the camera to the far clipping plane

void ***setSpeed****(float dist)*

sets the camera speed, you will need to play around with this to get is right for your world-space.

void ***setPositionAndLookat(PVector pos, PVector lookat)***

Sets the position of the camera and the look at of the camera. The up-vector is defaulted to (0,1,0)

void ***startDrawHUD*** *()*

call this before any 2D drawing, such as a HUD, but after you have finished drawing your 3D scene (otherwise those 3D things drawn later will be on top of your HUD)

void ***endDrawHUD*** *()*

call this to end any 2D drawing

SimRay ***getWindowRay(****PVector winPos****)*** *()*

Returns a ray directly into the window at position winPos.

SimRay ***getMouseRay()*** *()*

Returns a ray directly into the window at the current mouse position. Useful for mouse-picking of objects in the scene

The default “flying camera” mode of the SimCamera uses the following keys and mouse operation

- Swivel around using the right-mouse-button
- Strafe using the Arrow-keys
- W and S to move forwards and backwards.

Probably no need to look beyond this to use the SimObject library

SimTriangle methods

This class is used to contain a 3D triangle. It is NOT derived from SimShape, so can only be used to store finally transformed vertices.

Constructors

SimTriangle()

Returns a “zero” triangle. For the moment just use the member variables p1,p2 and p3 to set the vertices.

Public member variable

*PVector **p1, p2, p3**;*

The three vertices of the triangle

Methods

*void **flip**()*

Reverses the winding of the triangle from CW to CCW or via versa.

*PVector **surfaceNormal**()*

Returns the surface normal of the triangle