Лабораторная работа №4:
Обнаружение пересечений

Цель:

Целью данной лабораторной работы является изучение методов обнаружения пересечения объектов в трёхмерной системе координат и организация взаимодействия с ними.

Справка: низко полигональные трёхмерные модели: http://tf3dm.com/3d-models/architecture
Библиотека интерфейса DAT GUI: https://code.google.com/p/dat-gui/

Часть 1: Кисть

Наглядным примером механизма обнаружения пересечений в сцене на основе лучей, является задача построения трёхмерной модели ландшафта при помощи манипулятора типа мышь.

Для обработки событий генерируемых мышью, вам могут понадобиться следующие вызовы:

```
renderer.domElement.addEventListener('mousedown',onDocumentMouseDown,false);
renderer.domElement.addEventListener('mouseup',onDocumentMouseUp,false);
renderer.domElement.addEventListener('mousemove',onDocumentMouseMove,false);
renderer.domElement.addEventListener('wheel',onDocumentMouseScroll,false);

function onDocumentMouseScroll( event ) {…}
function onDocumentMouseMove( event )   {…}
function onDocumentMouseDown( event )   {…}
function onDocumentMouseUp( event )   {…}
```

Как можно понять из названий, приведённые обработчики отвечают за события:

`mousedown` – нажатие кнопки мыши, генерируется в момент нажатия на кнопку мыши

`mouseup` – нажатие кнопки мыши, генерируется в момент отжатия кнопки мыши

`mousemove` – нажатие кнопки мыши, генерируется в момент перемещения мыши

`wheel` – нажатие кнопки мыши, генерируется в момент вращения колёсика мыши

Обработчик каждого из этих событий, принимает в качестве параметра объект `event`, содержащий информацию о текущем состоянии устройства. В процессе выполнения лабораторной работы вам могут понадобиться следующие поля:

`event.wheelDelta` – смещение колеса мыши. Может быть положительным или отрицательным.

`event.clientX/ event.clientY` – положение курсора мыши в экранных координатах.

`event.which` – номер нажатой кнопки мыши.

Для того что бы отключить всплывающее меню по нажатию правой кнопки мыши на странице, можно воспользоваться следующим обработчиком события:

```
renderer.domElement.addEventListener("contextmenu",
                        function (event)
                        {
                            event.preventDefault();
                        });
```

Обнаружить пересечение курсора мыши с плоскостью можно следующим образом:

```
var mouse = { x: 0, y: 0 }; //переменная для хранения координат мыши
//массив для объектов, проверяемых на пересечение с курсором
var targetList = [];

…
//добавление в массив плоскость (ландшафт)
targetList.push(mesh);
…
//определение позиции мыши
mouse.x = ( event.clientX / window.innerWidth ) * 2 - 1;
mouse.y = -( event.clientY / window.innerHeight ) * 2 + 1;

//создание луча, исходящего из позиции камеры и проходящего сквозь позицию курсора мыши
var vector = new THREE.Vector3( mouse.x, mouse.y, 1 );
vector.unproject(camera);

var ray = new THREE.Raycaster( camera.position,
                               vector.sub( camera.position ).normalize() );

// создание массива для хранения объектов, с которыми пересечётся луч
var intersects = ray.intersectObjects( targetList );

// если луч пересёк какой-либо объект из списка targetList
if ( intersects.length > 0 )
{
    //печать списка полей объекта
    console.log(intersects[0]);
}
```

При выполнении лабораторной работы, вам могут понадобиться следующие поля объекта, возвращаемого функцией `ray.intersectObjects`:

`point` – координаты точки пересечения луча с объектом.

`object` – ссылка на объект, с которым произошло пересечение.

`face` – ссылка на треугольник с которым произошло пересечение.

`faceIndex` – индекс треугольника с которым произошло пересечение.

`distance` – дистанция до объекта, с которым произошло пересечение.

Теперь, умея обрабатывать события мыши и определять место пересечения курсора и плоскости, вы можете создать объект “кисть”.

Под кистью понимается объект трёхмерного мира, обозначающий зону на плоскости, к которой будут применены некие изменения. Существует множество способов обозначить такую область, например стандартные объекты типа цилиндр и круг.

Для того что бы добавить объект типа цилиндр, можно использовать следующий код:

```
//параметры цилиндра: диаметр вершины, диаметр основания, высота, число сегментов
var geometry = new THREE.CylinderGeometry( 1.5, 0, 5, 64 );
var cyMaterial = new THREE.MeshLambertMaterial( {color: 0x888888} );
var cylinder = new THREE.Mesh( geometry, cyMaterial );
scene.add( cylinder );
```

Для создания окружности, рекомендуется использовать функцию, аналогичную функции отображения орбит планет, из лабораторной работы номер 2.

Изменять геометрию объекта можно получив структуру, содержащую массив вершин этого объекта:

```
var vertices = geometry.getAttribute("position"); //получение массива вершин плоскости

for (var i = 0; i < vertices.array.length; i+=3) //перебор вершин
{
    var x = vertices.array[i];              //получение координат вершин по X
    var y = vertices.array[i+1];            //получение координат вершин по Y
    var z = vertices.array[i+2];            //получение координат вершин по Z
    vertices.array[i+1] += Math.sin(x*z);  //изменение координат по Y
}
geometry.setAttribute( 'position', vertices );   //установка изменённых вершин
```

После внесения изменений в геометрию плоскости, следует использовать команды:

```
geometry.computeVertexNormals();                  //пересчёт нормалей
geometry.attributes.position.needsUpdate = true; //обновление вершин
geometry.attributes.normal.needsUpdate = true;   //обновление нормалей
```

Задание: реализовать программу, поддерживающую следующий функционал:

- вывод на экран плоскости, с наложенной на неё текстурой (см. задание к части 3, лабораторной работы 1)
- вращение камеры вокруг выведенной плоскости
- отрисовка “кисти”, при наведении курсора мыши на сетку и возможность изменения её радиуса при помощи “колеса” мыши
- возможность изменения высот вершин плоскости при помощи нажатия кнопок мыши

Справка: в примере, для изменения высот ландшафта в пределах радиуса кисти была использована формула сферы:

$$h = \sqrt{r^2 - ((x_2 - x_1)^2 + (z_2 - z_1)^2)}$$

Где:
h - высота в текущей точки
r – радиус сферы
x_1/z_1 – центр сферы
x_2/z_2 – координаты текущей точки

Формула описывает: насколько нужно поднять каждую вершину внутри радиуса “кисти”, для того что бы получить полусферу на плоскости с радиусом в центре “кисти”.

Справка: Для того, чтобы определить координаты точек окружности в мировой системе координат можно использовать следующий код:

```
for (var i = 0; i < circle.geometry.attributes.position.array.length-1; i+=3)
{
    //получение позиции в локальной системе координат
    var pos = new THREE.Vector3();
    pos.x = circle.geometry.attributes.position.array[i];
    pos.y = circle.geometry.attributes.position.array[i+1];
    pos.z = circle.geometry.attributes.position.array[i+2];
    //нахождение позиции в глобальной системе координат
    pos.applyMatrix4(circle.matrixWorld);
}
```

Снимок экрана с примером работы программы:

Часть 2: Интерфейс

Для продолжения выполнения лабораторной работы необходимо организовать добавление объектов в сцену, с возможностью их перемещения, вращения и масштабирования. Это удобно сделать при помощи библиотеки графического интерфейса пользователя DAT GUI.

Работа с интерфейсом выглядит следующим образом:

```
//объект интерфейса и его ширина
var gui = new dat.GUI();
gui.width = 200;

//массив переменных, ассоциированных с интерфейсом
var params =
{
    sx: 0, sy: 0, sz: 0,
    brush: false,
    addHouse: function() { addMesh() },
    del: function() { delMesh() }
};
//создание вкладки
var folder1 = gui.addFolder('Scale');

//ассоциирование переменных отвечающих за масштабирование
//в окне интерфейса они будут представлены в виде слайдера
//минимальное значение - 1, максимальное – 100, шаг – 1
//listen означает, что изменение переменных будет отслеживаться
var meshSX = folder1.add( params, 'sx' ).min(1).max(100).step(1).listen();
var meshSY = folder1.add( params, 'sy' ).min(1).max(100).step(1).listen();
var meshSZ = folder1.add( params, 'sz' ).min(1).max(100).step(1).listen();
//при запуске программы папка будет открыта
folder1.open();
//описание действий совершаемых при изменении ассоциированных значений
```

```
meshSX.onChange(function(value) {…});
meshSY.onChange(function(value)  {…});
meshSZ.onChange(function(value) {…});

//добавление чек бокса с именем brush
var cubeVisible = gui.add( params, 'brush' ).name('brush').listen();
cubeVisible.onChange(function(value)
{
    // value принимает значения true и false
});

//добавление кнопок, при нажатии которых будут вызываться функции addMesh
//и delMesh соответственно. Функции описываются самостоятельно.
gui.add( params, 'addHouse' ).name( "add house" );
gui.add( params, 'del' ).name( "delete" );

//при запуске программы интерфейс будет раскрыт
gui.open();
```

Изменения вносимые в поля массива `params`, будут отражаться на ассоциированных с ними полях интерфейса.

Ссылки на добавляемые в сцену объекты желательно хранить в отдельном массиве, который будет использоваться для обнаружения пересечения курсора мыши и объектов в сцене. Сам алгоритм обнаружения пересечения выглядит практически так же как использованный в предыдущей части, за исключением:

```
var intersects = ray.intersectObjects( draworder, true );
```

где первый параметр, это массив объектов, а второй, команда на рекурсивный поиск пересечений. Рекурсивный поиск пересечений нужен для того, что бы определять пересечения с геометриями, входящими в подъобъекты объектов сцены.

Для того, что бы получится ссылку на объект, найдя пересечение с его подмножеством, следует использовать свойство подобъектовобъектов `child.parent`.

Удаление объекта из массива и сцены может выглядеть следующим образом:

```
//поиск индекса эллемента link в массиве draworder
var ind = draworder.indexOf(link);
//если такой индекс существует, удаление одного эллемента из массива
if (~ind) draworder.splice(ind, 1);

//удаление из сцены объекта, на который ссылается link
scene.remove(link);
```

где `link` – ссылка на объект который необходимо удалить. Получить её можно через поиск пересечения курсора с объектами в сцене.

Задание: реализовать программу, поддерживающую следующий функционал:

- вывод на экран интерфейса при помощи библиотеки DAT GUI. Интерфейс должен поддерживать возможности по изменению масштаба и углов поворота моделей в сцене,

добавление и удаление моделей в сцену, и переключение режимов курсора между “кистью” и курсором выбора объектов

- выбор и перемещение объектов в сцене при помощи курсора мыши.
- добавление и удаление объектов в сцену

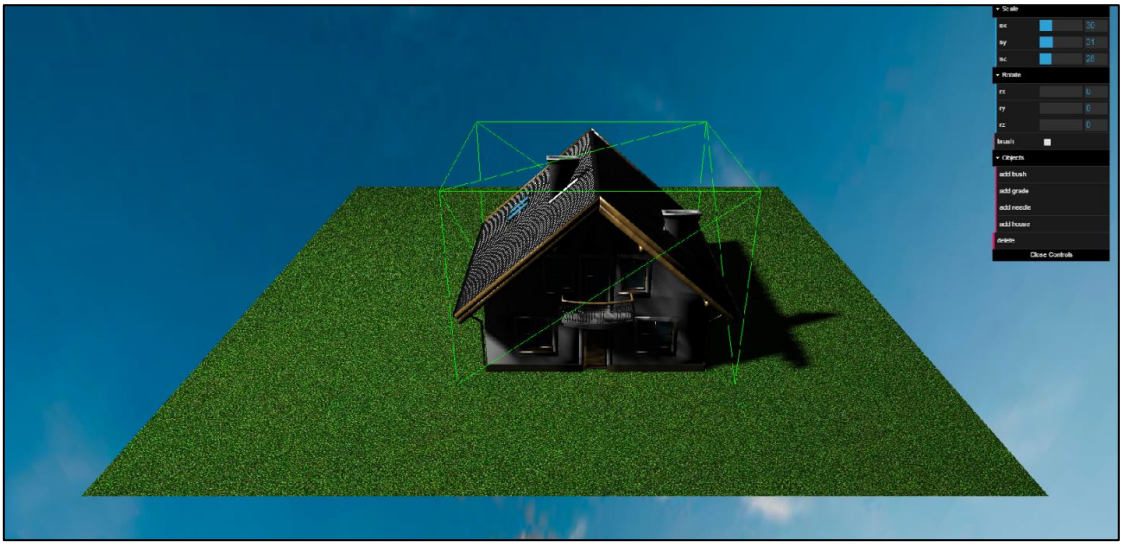
Скриншот работы программы:

Часть 3: Ограничивающие объёмы

На данном этапе, пересечение луча исходящего из позиции курсора в сцене считается для каждого треугольника модели. Подобный подход может быть использован и для поиска пересечения объектов в сцене. Лучи строится из геометрического центра модели до каждой из её вершин, а затем, каждый треугольник каждой модели проверяется на пересечение с каждым из этих лучей. Очевидно, подобный подход будет иметь избыточную трудоёмкость.

Для того, чтобы сократить вычисления на поиск пересечений в сцене, используются ограничивающие объёмы – примитивы (параллелепипед, сфера, цилиндр), содержащие в себе модель или её подмножество.

Простейшим примером такого объёма является AABB, ограничивающая коробка, выровненная по осям координат.



Для создания AABB в используемой библиотеке существует специальный класс Box3. Для хранения AABB удобно использовать поле объектов типа Object3D – userData:

```
//создание объекта Box3 и установка его вокруг объекта object
object.userData.bbox = new THREE.Box3();
object.userData.bbox.setFromObject(object);
```

объекты типа Box3 содержат в себе `boundingBox.min` и `boundingBox.max`, максимальное и минимальное значения AABB по трём осям координат.

Для проверки пересечения двух AABB используется функция:

```
object.userData.bbox.IntersectsBox(box);
```

Следует заметить, что перед проверкой на пересечение необходимо установить позицию AABB при помощи, указанной выше функции, поскольку объект типа Box3 хранит диагональ AABB в мировых координатах без преобразований.

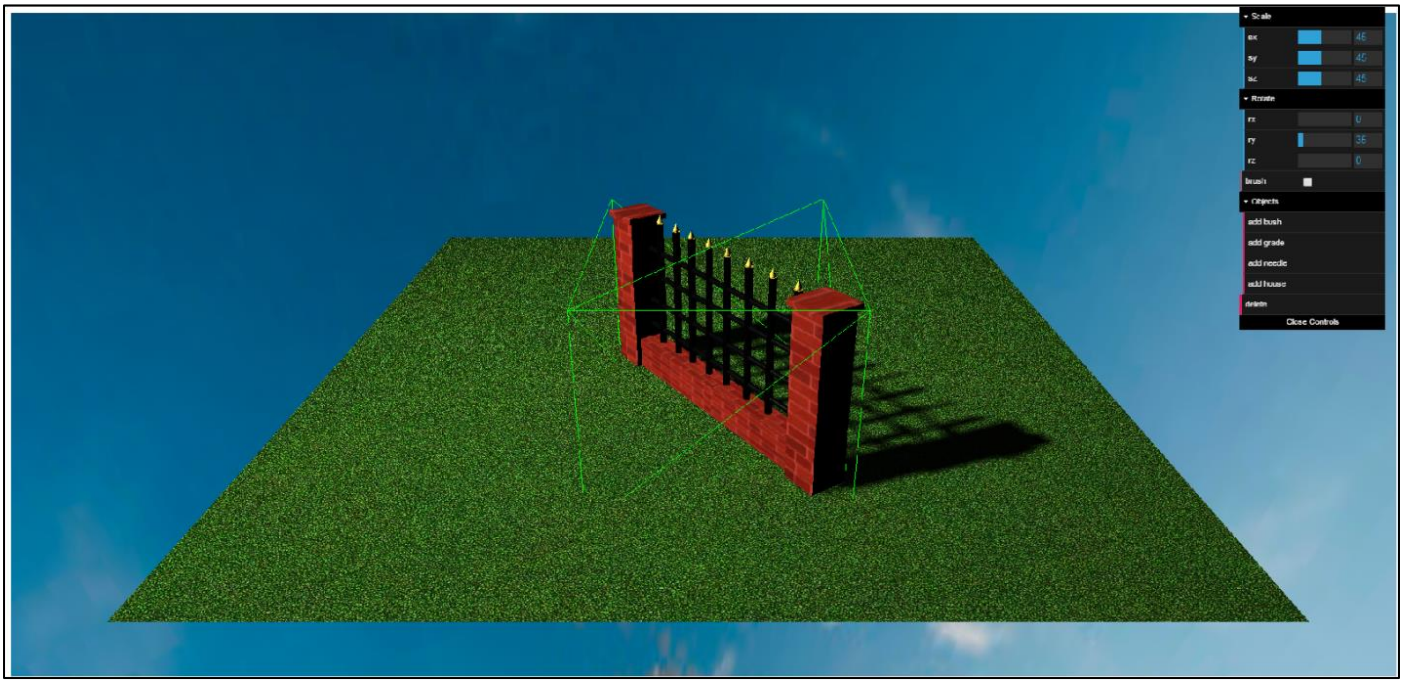
Для визуализации AABB, может быть использован стандартный объект, содержащий в себе Box3:

```
//создание объекта, содержащего Box3 и модель параллелепипеда
object.userData.box = new THREE.BoxHelper( object, 0xffff00 );
//обновление размеров коробки
object.userData.box.update();
//добавление коробки в сцену
scene.add( object.userData.box );
```

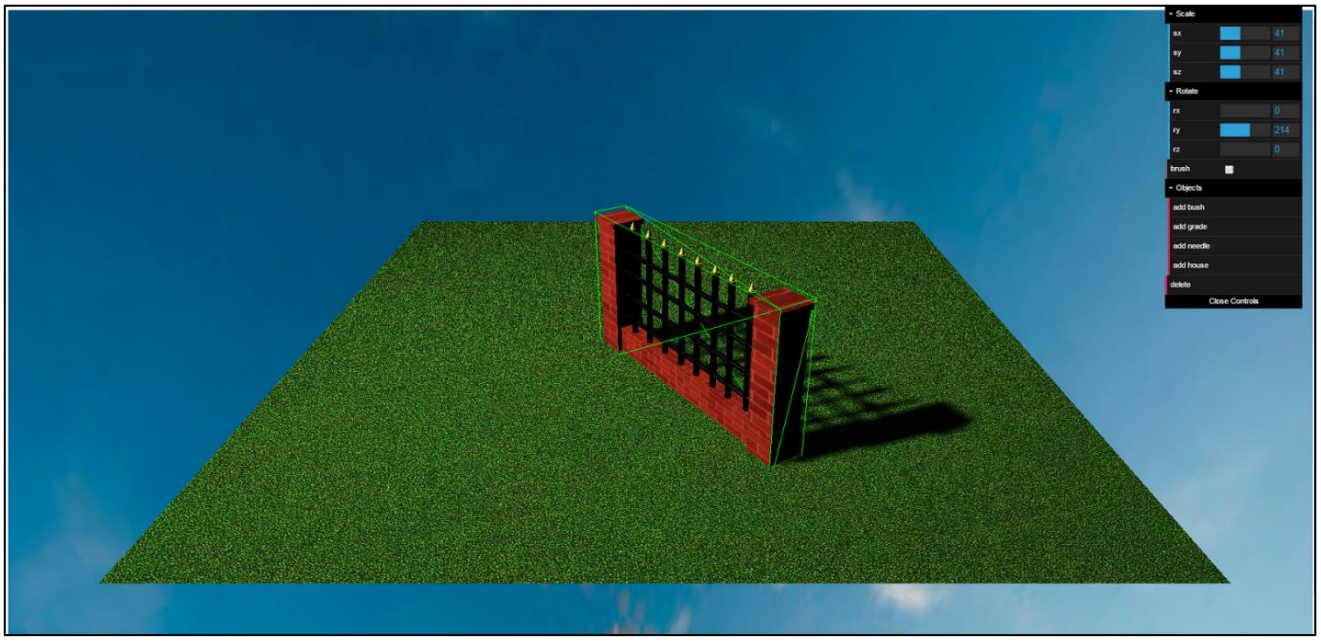
Для того, чтобы скрыть или отобразить коробку, можно использовать следующий код:

```
//скрытие объекта
object.userData.box.material.visible = false;
//отмена скрытия объекта
object.userData.box.material.visible = true;
```

Недостатком подобного ограничивающего объёма является не оптимальный охват объекта:



Для более точной проверки пересечений, используется OBB, ориентированная ограничивающая коробка:



OOB представляет собой обычный параллелепипед описанный вокруг объекта, и содержащий все его преобразования.

Создать OOB можно при помощи стандартного объекта BoxGeometry:

```
var geometry = new THREE.BoxGeometry( 1, 1, 1 );
var material = new THREE.MeshBasicMaterial(
    { color: 0x00ff00, wireframe: true }
);
object.userData.cube = new THREE.Mesh( geometry, material );
scene.add( object.userData.cube );
```

после чего, необходимо выставить размер, позицию, поворот и масштабирование коробки так, чтобы описать её вокруг объекта.

Пример:

```
//создание объекта Box3 и установка его вокруг объекта object
object.userData.bbox = new THREE.Box3();
object.userData.bbox.setFromObject(object);

//создание куба единичного размера
var geometry = new THREE.BoxGeometry( 1, 1, 1 );
var material = new THREE.MeshBasicMaterial(
    { color: 0x00ff00, wireframe: true }
);

object.userData.cube = new THREE.Mesh( geometry, material );
scene.add( object.userData.cube );

//получение позиции центра объекта
var pos = new THREE.Vector3();
object.userData.bbox.getCenter(pos);
```

```
//получение размеров объекта
var size = new THREE.Vector3();
object.userData.bbox.getSize(size);

//установка позиции и размера объекта в куб
object.userData.cube.position.copy(pos);
object.userData.cube.scale.set(size.x, size.y, size.z);
```

Проверить пересечение двух OBB можно при помощи функции intersect, описанной в приложении ниже.

Задание: добавить в разработанную в предыдущих частях программу следующий функционал:

- расчёт и хранение AABB для добавляемых в сцену объектов
- расчёт и хранение OBB для добавляемых в сцену объектов
- выбор объектов в сцене при помощи проверки пересечения с их OBB
- визуализацию OBB выбранного объекта
- двухуровневую проверку пересечения объектов в сцене, в случае если обнаружено пересечение объектов, передвигаемый объект должен устанавливаться в “старую” позицию (до обнаружения пересечения)

Приложение: функция intersect

Структура, представляющая OBB:

```
var obb = {};
//структура состоит из матрицы поворота, позиции и половины размера
obb.basis = new THREE.Matrix4();
obb.halfSize = new THREE.Vector3();
obb.position = new THREE.Vector3();

//получение позиции центра объекта
object.userData.bbox.getCenter(obb.position);
//получение размеров объекта
object.userData.bbox.getSize(obb.halfSize).multiplyScalar(0.5);
//получение матрицы поворота объекта
obb.basis.extractRotation(object.matrixWorld);

//структура хранится в поле userData объекта
object.userData.obb = obb;
```

Пример использования функции:

```
//перебор всех OBB объектов сцены
for(var i = 0; i < targets.length; i++)
{
    if (targets[i] !== picked)
    {
        targets[i].material.visible = false;
        intr = intersect(picked.userData, targets[i].userData);

        //объект пересечение с которым было обнаружено
        //становится видимым
        if (intr)
        {
            targets[i].material.visible = true;
            break;
        }
    }
}
```

Реализация функции:

```
//оригинал алгоритма и реализацию класса OBB можно найти по ссылке:
//https://github.com/Mugen87/yume/blob/master/src/javascript/engine/etc/OBB.js
function intersect(ob1, ob2)
{
    var xAxisA = new THREE.Vector3();

    var yAxisA = new THREE.Vector3();
    var zAxisA = new THREE.Vector3();

    var xAxisB = new THREE.Vector3();
    var yAxisB = new THREE.Vector3();
```

```
    var zAxisB = new THREE.Vector3();

    var translation = new THREE.Vector3();

    var vector = new THREE.Vector3();

    var axisA = [];
    var axisB = [];
    var rotationMatrix = [ [], [], [] ];
    var rotationMatrixAbs = [ [], [], [] ];

    var _EPSILON = 1e-3;

    var halfSizeA, halfSizeB;
    var t, i;

    ob1.obb.basis.extractBasis( xAxisA, yAxisA, zAxisA );
    ob2.obb.basis.extractBasis( xAxisB, yAxisB, zAxisB );

    // push basis vectors into arrays, so you can access them via indices
    axisA.push( xAxisA, yAxisA, zAxisA );
    axisB.push( xAxisB, yAxisB, zAxisB );

    // get displacement vector
    vector.subVectors( ob2.obb.position, ob1.obb.position );
    // express the translation vector in the coordinate frame of the current
    // OBB (this)
    for ( i = 0; i < 3; i++ )
    {
    translation.setComponent( i, vector.dot( axisA[ i ] ) );
    }

    // generate a rotation matrix that transforms from world space to the
    // OBB's coordinate space
    for ( i = 0; i < 3; i++ )
    {
    for ( var j = 0; j < 3; j++ )
    {
            rotationMatrix[ i ][ j ] = axisA[ i ].dot( axisB[ j ] );
            rotationMatrixAbs[ i ][ j ] = Math.abs( rotationMatrix[ i ][ j ] ) + _EPSILON;
    }
    }

    // test the three major axes of this OBB
    for ( i = 0; i < 3; i++ )
    {
        vector.set( rotationMatrixAbs[ i ][ 0 ], rotationMatrixAbs[ i ][ 1 ], rotationMatrixAbs[ i ][ 2 ]
);

    halfSizeA = ob1.obb.halfSize.getComponent( i );
    halfSizeB = ob2.obb.halfSize.dot( vector );
```

```
    if ( Math.abs( translation.getComponent( i ) ) > halfSizeA + halfSizeB )
    {
            return false;
    }
    }

    // test the three major axes of other OBB
    for ( i = 0; i < 3; i++ )
    {
    vector.set( rotationMatrixAbs[ 0 ][ i ], rotationMatrixAbs[ 1 ][ i ], rotationMatrixAbs[ 2 ][ i ] );

    halfSizeA = ob1.obb.halfSize.dot( vector );
    halfSizeB = ob2.obb.halfSize.getComponent( i );

    vector.set( rotationMatrix[ 0 ][ i ], rotationMatrix[ 1 ][ i ], rotationMatrix[ 2 ][ i ] );
    t = translation.dot( vector );

    if ( Math.abs( t ) > halfSizeA + halfSizeB )
    {
            return false;
    }
    }

    // test the 9 different cross-axes

    // A.x <cross> B.x
    halfSizeA = ob1.obb.halfSize.y * rotationMatrixAbs[ 2 ][ 0 ] + ob1.obb.halfSize.z *
rotationMatrixAbs[ 1 ][ 0 ];
    halfSizeB = ob2.obb.halfSize.y * rotationMatrixAbs[ 0 ][ 2 ] + ob2.obb.halfSize.z *
rotationMatrixAbs[ 0 ][ 1 ];

    t = translation.z * rotationMatrix[ 1 ][ 0 ] - translation.y * rotationMatrix[ 2 ][ 0 ];

    if ( Math.abs( t ) > halfSizeA + halfSizeB )
    {
    return false;
    }

    // A.x < cross> B.y
    halfSizeA = ob1.obb.halfSize.y * rotationMatrixAbs[ 2 ][ 1 ] + ob1.obb.halfSize.z *
rotationMatrixAbs[ 1 ][ 1 ];
    halfSizeB = ob2.obb.halfSize.x * rotationMatrixAbs[ 0 ][ 2 ] + ob2.obb.halfSize.z *
rotationMatrixAbs[ 0 ][ 0 ];

    t = translation.z * rotationMatrix[ 1 ][ 1 ] - translation.y * rotationMatrix[ 2 ][ 1 ];

    if ( Math.abs( t ) > halfSizeA + halfSizeB )
    {
        return false;
    }
```

```
    // A.x <cross> B.z
    halfSizeA = ob1.obb.halfSize.y * rotationMatrixAbs[ 2 ][ 2 ] + ob1.obb.halfSize.z *
rotationMatrixAbs[ 1 ][ 2 ];
    halfSizeB = ob2.obb.halfSize.x * rotationMatrixAbs[ 0 ][ 1 ] + ob2.obb.halfSize.y *
rotationMatrixAbs[ 0 ][ 0 ];

    t = translation.z * rotationMatrix[ 1 ][ 2 ] - translation.y * rotationMatrix[ 2 ][ 2 ];

    if ( Math.abs( t ) > halfSizeA + halfSizeB )
    {
        return false;
    }

    // A.y <cross> B.x
    halfSizeA = ob1.obb.halfSize.x * rotationMatrixAbs[ 2 ][ 0 ] + ob1.obb.halfSize.z *
rotationMatrixAbs[ 0 ][ 0 ];
    halfSizeB = ob2.obb.halfSize.y * rotationMatrixAbs[ 1 ][ 2 ] + ob2.obb.halfSize.z *
rotationMatrixAbs[ 1 ][ 1 ];

    t = translation.x * rotationMatrix[ 2 ][ 0 ] - translation.z * rotationMatrix[ 0 ][ 0 ];

    if ( Math.abs( t ) > halfSizeA + halfSizeB )
    {
        return false;
    }

    // A.y <cross> B.y
    halfSizeA = ob1.obb.halfSize.x * rotationMatrixAbs[ 2 ][ 1 ] + ob1.obb.halfSize.z *
rotationMatrixAbs[ 0 ][ 1 ];
    halfSizeB = ob2.obb.halfSize.x * rotationMatrixAbs[ 1 ][ 2 ] + ob2.obb.halfSize.z *
rotationMatrixAbs[ 1 ][ 0 ];

    t = translation.x * rotationMatrix[ 2 ][ 1 ] - translation.z * rotationMatrix[ 0 ][ 1 ];

    if ( Math.abs( t ) > halfSizeA + halfSizeB )
    {
    return false;
    }

    // A.y <cross> B.z
    halfSizeA = ob1.obb.halfSize.x * rotationMatrixAbs[ 2 ][ 2 ] + ob1.obb.halfSize.z *
rotationMatrixAbs[ 0 ][ 2 ];
    halfSizeB = ob2.obb.halfSize.x * rotationMatrixAbs[ 1 ][ 1 ] + ob2.obb.halfSize.y *
rotationMatrixAbs[ 1 ][ 0 ];

    t = translation.x * rotationMatrix[ 2 ][ 2 ] - translation.z * rotationMatrix[ 0 ][ 2 ];

    if ( Math.abs( t ) > halfSizeA + halfSizeB )
    {
    return false;
    }
```

```
    // A.z <cross> B.x
    halfSizeA = ob1.obb.halfSize.x * rotationMatrixAbs[ 1 ][ 0 ] + ob1.obb.halfSize.y *
rotationMatrixAbs[ 0 ][ 0 ];
    halfSizeB = ob2.obb.halfSize.y * rotationMatrixAbs[ 2 ][ 2 ] + ob2.obb.halfSize.z *
rotationMatrixAbs[ 2 ][ 1 ];

    t = translation.y * rotationMatrix[ 0 ][ 0 ] - translation.x * rotationMatrix[ 1 ][ 0 ];

    if ( Math.abs( t ) > halfSizeA + halfSizeB )
    {
    return false;
    }

    // A.z <cross> B.y
    halfSizeA = ob1.obb.halfSize.x * rotationMatrixAbs[ 1 ][ 1 ] + ob1.obb.halfSize.y *
rotationMatrixAbs[ 0 ][ 1 ];
    halfSizeB = ob2.obb.halfSize.x * rotationMatrixAbs[ 2 ][ 2 ] + ob2.obb.halfSize.z *
rotationMatrixAbs[ 2 ][ 0 ];

    t = translation.y * rotationMatrix[ 0 ][ 1 ] - translation.x * rotationMatrix[ 1 ][ 1 ];

    if ( Math.abs( t ) > halfSizeA + halfSizeB )
    {
    return false;
    }

    // A.z <cross> B.z
    halfSizeA = ob1.obb.halfSize.x * rotationMatrixAbs[ 1 ][ 2 ] + ob1.obb.halfSize.y *
rotationMatrixAbs[ 0 ][ 2 ];
    halfSizeB = ob2.obb.halfSize.x * rotationMatrixAbs[ 2 ][ 1 ] + ob2.obb.halfSize.y *
rotationMatrixAbs[ 2 ][ 0 ];

    t = translation.y * rotationMatrix[ 0 ][ 2 ] - translation.x * rotationMatrix[ 1 ][ 2 ];

    if ( Math.abs( t ) > halfSizeA + halfSizeB )
    {
    return false;
    }

    // no separating axis exists, so the two OBB don't intersect
    return true;
}
```