# Volumes englobants

Gérer les collisions entre les objets du monde est essentiel en jeu vidéo. Toutefois les calculs de collisions peuvent coûter cher et il faut donc faire un choix en fonction du type de jeu. Pour une physique très poussée, il faudra surement une système de tests de collision avec lancés de rayons. Toutefois pour la plupart des cas, un test simplifié suffit, en utilisant les bounding volumes.
Les bounding spheres et les bounding boxes sont les plus utilisés car les plus simples en calculs. Même pour des collisions plus précises, on peut éliminer un grand nombre d'objets de la scène avant de tester les collisions entre les rayons et les triangles des objets non-éliminés.
Au cours de ce TD, nous verrons l'implémentation de plusieurs types de Bounding Volumes. Les collisions entre Bounding Volumes seront vus dans un autre TD.

Le projet qui vous est donné reprend la classe Game ainsi que la classe MeshLoader que vous aviez utilisé dans le TD6.
Dans ce projet vous pouvez afficher les differents volumes englobants en appuyant sur espace. Vous pouvez voir qu'une bounding sphere et une bounding box ont étés implémentées, néanmoins elles ne sont pas calculées en fonction du mesh.

## Instructions :

1) Dans la classe BoundingSphere ajoutez une fonction "ComputeBoundingSphere" qui sera appelée dans le constructeur et qui calculera le centre et le rayon de la bounding sphère en fonction des vertexs du mesh.

2) Dans la classe OBB ajoutez une fonction "ComputeOBoundingBox" qui sera appelé dans le constructeur et qui calculera la taille de la box en fonction des vertexs du mesh.

3) Créez une classe AABB qui héritera de la classe Cuboid que vous avez utilisée auparavant et qui calculeras les vertices de la box en fonction des vertices du mesh. La box changeant de forme en fonction du placement du mesh dans l'espace, la fonction de calcul des vertice devra cette fois-ci être appelée à chaque rendu de la box.

## Informations :

Pour la bounding sphere :

*Centre du mesh en X* = (Xmax + Xmin) / 2
*Calcul du rayon* :

```
float Radius ← 0
Pour i allant de 0 au nombre de vertice du mesh Faire
    float Distance ← norme au carré (vertice ni – centre du mesh)
    Si Distance > Radius Alors Radius ← Distance
    Fin Si
Fin Pour
Radius ← racine carré(Radius)
```

Vous pouvez vous aider avec le site : http://www.geometrictools.com

# Documentation :

## D3DXCreateSphere Function

Uses a left-handed coordinate system to create a mesh containing a sphere.

**Syntax :**

```
HRESULT  D3DXCreateSphere(
  __in   LPDIRECT3DDEVICE9 pDevice,
  __in   FLOAT Radius,
  __in   UINT Slices,
  __in   UINT Stacks,
  __out  LPD3DXMESH *ppMesh,
  __out  LPD3DXBUFFER *ppAdjacency
);
```

**Parameters :**

*pDevice* [in]
**LPDIRECT3DDEVICE9**

Pointer to an **IDirect3DDevice9** interface, representing the device associated with the created sphere mesh.

*Radius* [in]
**FLOAT**

Radius of the sphere. This value should be greater than or equal to 0.0f.

*Slices* [in]
**UINT**

Number of slices about the main axis.

*Stacks* [in]
**UINT**

Number of stacks along the main axis.

*ppMesh* [out]
**LPD3DXMESH**

Address of a pointer to the output shape, an **ID3DXMesh** interface.

*ppAdjacency* [out]
**LPD3DXBUFFER**

Address of a pointer to an **ID3DXBuffer** interface. When the method returns, this parameter is filled with an array of three DWORDs per face that specify the three neighbors for each face in the mesh. NULL can be specified.

**Return Value :**

**HRESULT**

If the function succeeds, the return value is D3D_OK. If the function fails, the return value can be one of the following: D3DERR_INVALIDCALL, D3DXERR_INVALIDDATA, E_OUTOFMEMORY.

## D3DXCreateBox Function

Uses a left-handed coordinate system to create a mesh containing an axis-aligned box.

**Syntax :**

```
HRESULT  D3DXCreateBox(
  __in   LPDIRECT3DDEVICE9 pDevice,
  __in   FLOAT Width,
  __in   FLOAT Height,
  __in   FLOAT Depth,
  __out  LPD3DXMESH *ppMesh,
  __out  LPD3DXBUFFER *ppAdjacency
);
```

**Parameters :**

*pDevice* [in]
**LPDIRECT3DDEVICE9**

Pointer to an **IDirect3DDevice9** interface, representing the device associated with the created box mesh.

*Width* [in]
**FLOAT**

Width of the box, along the x-axis.

*Height* [in]
**FLOAT**

Height of the box, along the y-axis.

*Depth* [in]
**FLOAT**

Depth of the box, along the z-axis.

*ppMesh* [out]
**LPD3DXMESH**

Address of a pointer to the output shape, an **ID3DXMesh** interface.

*ppAdjacency* [out]
**LPD3DXBUFFER**

Address of a pointer to an **ID3DXBuffer** interface. When the method returns, this parameter is filled with an array of three DWORDs per face that specify the three neighbors for each face in the mesh. NULL can be specified.

**Return Value :**

**HRESULT**

If the function succeeds, the return value is D3D_OK. If the function fails, the return value can be one of the following: D3DERR_INVALIDCALL, D3DXERR_INVALIDDATA, E_OUTOFMEMORY.

## D3DXVec3LengthSq Function

Returns the square of the length of a 3D vector.

**Syntax :**

```
 FLOAT  D3DXVec3LengthSq(
  __in  const D3DXVECTOR3 *pV
);
```

**Parameters :**

*pV* [in]
**D3DXVECTOR3**

Pointer to the source **D3DXVECTOR3** structure.

**Return Value :**

**FLOAT**

The vector's squared length.

## ID3DXBaseMesh::LockVertexBuffer Method

Locks a vertex buffer and obtains a pointer to the vertex buffer memory.

**Syntax :**

```
HRESULT LockVertexBuffer(
  [in]           DWORD Flags,
  [out, retval]  LPVOID *ppData
);
```

**Parameters :**

*Flags* [in]
**DWORD**

Combination of zero or more locking flags that describe the type of lock to perform. For this method, the valid flags are:

- D3DLOCK_DISCARD
- D3DLOCK_NO_DIRTY_UPDATE

- D3DLOCK_NOSYSLOCK
- D3DLOCK_READONLY
- D3DLOCK_NOOVERWRITE

For a description of the flags, see D3DLOCK.

*ppData* [out]
**LPVOID**

VOID* pointer to a buffer containing the vertex data.

**Return Value :**

**HRESULT**

If the method succeeds, the return value is D3D_OK. If the method fails, the return value can be D3DERR_INVALIDCALL.

## D3DXVec3TransformCoord Function

Transforms a 3D vector by a given matrix, projecting the result back into w = 1.

**Syntax :**

```
D3DXVECTOR3 * D3DXVec3TransformCoord(
  __inout  D3DXVECTOR3 *pOut,
  __in     const D3DXVECTOR3 *pV,
  __in     const D3DXMATRIX *pM
);
```

**Parameters :**

*pOut* [in, out]
**D3DXVECTOR3**

Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

*pV* [in]
**D3DXVECTOR3**

Pointer to the source **D3DXVECTOR3** structure.

*pM* [in]
**D3DXMATRIX**

Pointer to the source **D3DXMATRIX** structure.

**Return Value :**

**D3DXVECTOR3**

Pointer to a **D3DXVECTOR3** structure that is the transformed vector.

**Remarks :**

This function transforms the vector, *pV* (x, y, z, 1), by the matrix, *pM*, projecting the result back into w=1.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3TransformCoord** function can be used as a parameter for another function.

Source : http://msdn.microsoft.com/en-us/library