Tutorial: Frustum Culling

The 3D viewing area on the screen where everything is drawn to is called the viewing frustum. Everything that is inside the frustum will be rendered to the screen by the video card. Everything that is outside of the frustum the video card will examine and then discard during the rendering process.

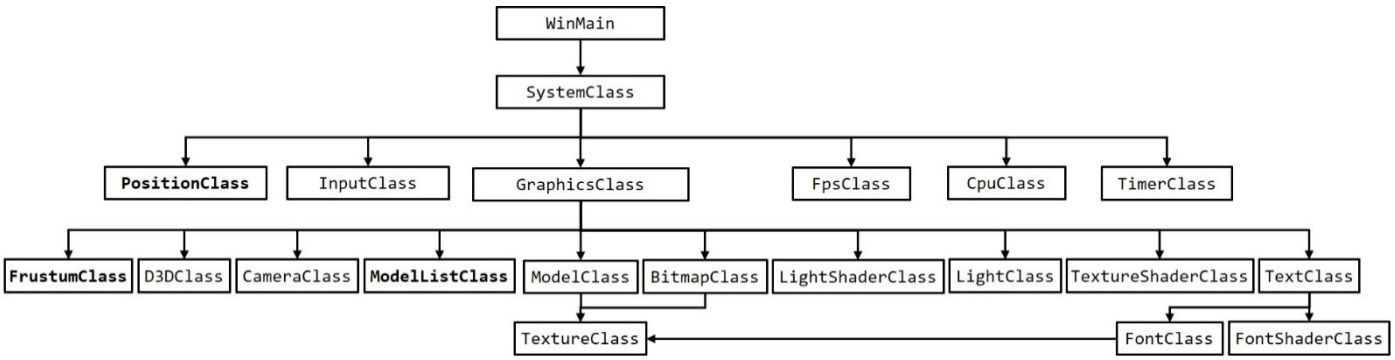
However, the process of depending on the video card to cull for us can be expensive if we have large scenes. For example, say we have a scene with 2000+ models that are 5,000 polygons each but only 10-20 are viewable at any given time. The video card has to examine every single triangle in all 2000 models to remove 1990 models from the scene, just so we can draw 10 models. As you can see this is very inefficient.

How frustum culling solves our problem is that we can instead determine before rendering if a model is in our frustum or not. This saves us sending all the triangles to the video card and allows us to just send the triangles that need to be drawn. How we do this is that we put either a cube, a rectangle, or a sphere around each model and just calculate if that cube, rectangle, or sphere is viewable. The math to do that is usually only a couple lines of code which then removes the need to possibly test several thousand triangles.

To demonstrate how this works we will first create a scene with 25 randomly placed spheres. We will then rotate the camera manually to test culling of the spheres that are out of our view using the left and right arrow keys. We will also use a counter and display the number of spheres that are being drawn and not culled for confirmation. We will use code from several of the previous tutorials to create the scene.

Framework

The frame work has mostly classes from several of the previous tutorials. We do have three new classes called FrustumClass, PositionClass, and ModelListClass. FrustumClass will encapsulate the frustum culling ability this tutorial is focused on. ModelListClass will contain a list of the position and color information of the 25 spheres that will be randomly generated each time we run the program. PositionClass will handle the viewing rotation of the camera based on if the user is pressing the left or right arrow key.



Frustumclass.h

The header file for the FrustumClass is fairly simple. The class doesn't require any initialization or shutdown. Each frame the ConstructFrustum function is called after the camera has first been rendered. The ConstructFrustum function uses the private m_planes to calculate and store the six planes of the view frustum based on the updated viewing location. From there we can call any of the four check functions to see if either a point, cube, sphere, or rectangle are inside the viewing frustum or not.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: frustumclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _FRUSTUMCLASS_H_
#define _FRUSTUMCLASS_H_

//////////////
// INCLUDES //
//////////////
#include <d3dx10math.h>


////////////////////////////////////////////////////////////////////////////////
```

```
// Class name: FrustumClass
//////////////////////////////////////////////////////////////////////////////
class FrustumClass
{
public:
	FrustumClass();
	FrustumClass(const FrustumClass&);
	~FrustumClass();

	void ConstructFrustum(float, D3DXMATRIX, D3DXMATRIX);

	bool CheckPoint(float, float, float);
	bool CheckCube(float, float, float, float);
	bool CheckSphere(float, float, float, float);
	bool CheckRectangle(float, float, float, float, float, float);

private:
	D3DXPLANE m_planes[6];
};

#endif
```

Frustumclass.cpp

```
//////////////////////////////////////////////////////////////////////////////
// Filename: frustumclass.cpp
//////////////////////////////////////////////////////////////////////////////
#include "frustumclass.h"

FrustumClass::FrustumClass()
{
}

FrustumClass::FrustumClass(const FrustumClass& other)
{
}

FrustumClass::~FrustumClass()
{
}
```

ConstructFrustum is called every frame by the GraphicsClass. It passes in the depth of the screen, the projection matrix, and the view matrix. We then use these input variables to calculate the matrix of the view frustum at that frame. With the new frustum matrix, we then calculate the six planes that form the view frustum.

```
void FrustumClass::ConstructFrustum(float screenDepth, D3DXMATRIX projectionMatrix, D3DXMATRIX viewMatrix)
{
	float zMinimum, r;
	D3DXMATRIX matrix;

	// Calculate the minimum Z distance in the frustum.
	zMinimum = -projectionMatrix._43 / projectionMatrix._33;
	r = screenDepth / (screenDepth - zMinimum);
	projectionMatrix._33 = r;
	projectionMatrix._43 = -r * zMinimum;

	// Create the frustum matrix from the view matrix and updated projection matrix.
	D3DXMatrixMultiply(&matrix, &viewMatrix, &projectionMatrix);

	// Calculate near plane of frustum.
	m_planes[0].a = matrix._14 + matrix._13;
	m_planes[0].b = matrix._24 + matrix._23;
	m_planes[0].c = matrix._34 + matrix._33;
	m_planes[0].d = matrix._44 + matrix._43;
	D3DXPlaneNormalize(&m_planes[0], &m_planes[0]);
```

```
	// Calculate far plane of frustum.
	m_planes[1].a = matrix._14 - matrix._13;
	m_planes[1].b = matrix._24 - matrix._23;
	m_planes[1].c = matrix._34 - matrix._33;
	m_planes[1].d = matrix._44 - matrix._43;
	D3DXPlaneNormalize(&m_planes[1], &m_planes[1]);

	// Calculate left plane of frustum.
	m_planes[2].a = matrix._14 + matrix._11;
	m_planes[2].b = matrix._24 + matrix._21;
	m_planes[2].c = matrix._34 + matrix._31;
	m_planes[2].d = matrix._44 + matrix._41;
	D3DXPlaneNormalize(&m_planes[2], &m_planes[2]);

	// Calculate right plane of frustum.
	m_planes[3].a = matrix._14 - matrix._11;
	m_planes[3].b = matrix._24 - matrix._21;
	m_planes[3].c = matrix._34 - matrix._31;
	m_planes[3].d = matrix._44 - matrix._41;
	D3DXPlaneNormalize(&m_planes[3], &m_planes[3]);

	// Calculate top plane of frustum.
	m_planes[4].a = matrix._14 - matrix._12;
	m_planes[4].b = matrix._24 - matrix._22;
	m_planes[4].c = matrix._34 - matrix._32;
	m_planes[4].d = matrix._44 - matrix._42;
	D3DXPlaneNormalize(&m_planes[4], &m_planes[4]);

	// Calculate bottom plane of frustum.
	m_planes[5].a = matrix._14 + matrix._12;
	m_planes[5].b = matrix._24 + matrix._22;
	m_planes[5].c = matrix._34 + matrix._32;
	m_planes[5].d = matrix._44 + matrix._42;
	D3DXPlaneNormalize(&m_planes[5], &m_planes[5]);

	return;
}
```

CheckPoint checks if a single point is inside the viewing frustum. This is the most general of the four checking algorithms but can be very efficient if used correctly in the right situation over the other checking methods. It takes the point and checks to see if it is inside all six planes. If the point is inside all six then it returns true, otherwise it returns false if not.

```
bool FrustumClass::CheckPoint(float x, float y, float z)
{
	int i;


	// Check if the point is inside all six planes of the view frustum.
	for(i=0; i<6; i++)
	{
		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3(x, y, z)) < 0.0f)
		{
			return false;
		}
	}

	return true;
}
```

CheckCube checks if any of the eight corner points of the cube are inside the viewing frustum. It only requires as input the center point of the cube and the radius, it uses those to calculate the 8 corner points of the cube. It then checks if any one of the corner points are inside all 6 planes of the viewing frustum. If it does find a point inside all six planes of the viewing frustum it returns true, otherwise it returns false.

```
bool FrustumClass::CheckCube(float xCenter, float yCenter, float zCenter, float radius)
{
	int i;

	// Check if any one point of the cube is in the view frustum.
	for(i=0; i<6; i++)
	{
		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter - radius), (yCenter - radius),
		(zCenter - radius))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter + radius), (yCenter - radius),
		(zCenter - radius))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter - radius), (yCenter + radius),
		(zCenter - radius))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter + radius), (yCenter + radius),
		(zCenter - radius))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter - radius), (yCenter - radius),
		(zCenter + radius))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter + radius), (yCenter - radius),
		(zCenter + radius))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter - radius), (yCenter + radius),
		(zCenter + radius))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter + radius), (yCenter + radius),
		(zCenter + radius))) >= 0.0f)
		{
			continue;
		}

		return false;
	}

	return true;
}
```

CheckSphere checks if the radius of the sphere from the center point is inside all six planes of the viewing frustum. If it is outside any of them then the sphere cannot be seen, and the function will return false. If it is inside all six the function returns true that the sphere can be seen.

```
bool FrustumClass::CheckSphere(float xCenter, float yCenter, float zCenter, float radius)
```

```
{
	int i;

	// Check if the radius of the sphere is inside the view frustum.
	for(i=0; i<6; i++)
	{
		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3(xCenter, yCenter, zCenter)) < -radius)
		{
			return false;
		}
	}

	return true;
}
```

CheckRectangle works the same as CheckCube except that that it takes as input the x radius, y radius, and z radius of the rectangle instead of just a single radius of a cube. It can then calculate the 8 corner points of the rectangle and do the frustum checks similar to the CheckCube function.

```
bool FrustumClass::CheckRectangle(float xCenter, float yCenter, float zCenter, float xSize, float ySize, float zSize)
{
	int i;

	// Check if any of the 6 planes of the rectangle are inside the view frustum.
	for(i=0; i<6; i++)
	{
		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter - xSize), (yCenter - ySize),
		(zCenter - zSize))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter + xSize), (yCenter - ySize),
		(zCenter - zSize))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter - xSize), (yCenter + ySize),
		(zCenter - zSize))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter - xSize), (yCenter - ySize),
		(zCenter + zSize))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter + xSize), (yCenter + ySize),
		(zCenter - zSize))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter + xSize), (yCenter - ySize),
		(zCenter + zSize))) >= 0.0f)
		{
			continue;
		}

		if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter - xSize), (yCenter + ySize),
		(zCenter + zSize))) >= 0.0f)
		{
			continue;
```

```
                }

                if(D3DXPlaneDotCoord(&m_planes[i], &D3DXVECTOR3((xCenter + xSize), (yCenter + ySize),
                (zCenter + zSize))) >= 0.0f)
                {
                        continue;
                }

                return false;
        }

        return true;
}
```

Modellistclass.h

ModelListClass is a new class for maintaining information about all the models in the scene. For this tutorial it only maintains the size and color of the sphere models since we only have one model type. This class can be expanded to maintain all the different types of models in the scene and indexes to their ModelClass, but I am keeping this tutorial simple for now.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: modellistclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _MODELLISTCLASS_H_
#define _MODELLISTCLASS_H_

///////////////
// INCLUDES //
///////////////
#include <d3dx10math.h>
#include <stdlib.h>
#include <time.h>

////////////////////////////////////////////////////////////////////////////////
// Class name: ModelListClass
////////////////////////////////////////////////////////////////////////////////
class ModelListClass
{
private:
        struct ModelInfoType
        {
                D3DXVECTOR4 color;
                float positionX, positionY, positionZ;
        };

public:
        ModelListClass();
        ModelListClass(const ModelListClass&);
        ~ModelListClass();

        bool Initialize(int);
        void Shutdown();

        int GetModelCount();
        void GetData(int, float&, float&, float&, D3DXVECTOR4&);

private:
        int m_modelCount;
        ModelInfoType* m_ModelInfoList;
};

#endif
```

Modellistclass.cpp

```
////////////////////////////////////////////////////////////////////////////////
// Filename: modellistclass.cpp
////////////////////////////////////////////////////////////////////////////////
#include "modellistclass.h"
```

The class constructor initializes the model information list to null.

```
ModelListClass::ModelListClass()
{
	m_ModelInfoList = 0;
}

ModelListClass::ModelListClass(const ModelListClass& other)
{
}

ModelListClass::~ModelListClass()
{
}

bool ModelListClass::Initialize(int numModels)
{
	int i;
	float red, green, blue;
```

First store the number of models that will be used and then create the list array of them using the ModelInfoType structure.

```
	// Store the number of models.
	m_modelCount = numModels;

	// Create a list array of the model information.
	m_ModelInfoList = new ModelInfoType[m_modelCount];
	if(!m_ModelInfoList)
	{
		return false;
	}
```

Seed the random number generator with the current time and then randomly generate the position of color of the models and store them in the list array.

```
	// Seed the random generator with the current time.
	srand((unsigned int)time(NULL));

	// Go through all the models and randomly generate the model color and position.
	for(i=0; i<m_modelCount; i++)
	{
		// Generate a random color for the model.
		red = (float)rand() / RAND_MAX;
		green = (float)rand() / RAND_MAX;
		blue = (float)rand() / RAND_MAX;

		m_ModelInfoList[i].color = D3DXVECTOR4(red, green, blue, 1.0f);

		// Generate a random position in front of the viewer for the mode.
		m_ModelInfoList[i].positionX = (((float)rand()-(float)rand())/RAND_MAX) * 10.0f;
		m_ModelInfoList[i].positionY = (((float)rand()-(float)rand())/RAND_MAX) * 10.0f;
		m_ModelInfoList[i].positionZ = ((((float)rand()-(float)rand())/RAND_MAX) * 10.0f) + 5.0f;
	}

	return true;
}
```

The Shutdown function releases the model information list array.

```
void ModelListClass::Shutdown()
{
	// Release the model information list.
	if(m_ModelInfoList)
	{
		delete [] m_ModelInfoList;
		m_ModelInfoList = 0;
	}

	return;
}
```

GetModelCount returns the number of models that this class maintains information about.

```
int ModelListClass::GetModelCount()
{
	return m_modelCount;
}
```

The GetData function extracts the position and color of a sphere at the given input index location.

```
void ModelListClass::GetData(int index, float& positionX, float& positionY, float& positionZ, D3DXVECTOR4& color)
{
	positionX = m_ModelInfoList[index].positionX;
	positionY = m_ModelInfoList[index].positionY;
	positionZ = m_ModelInfoList[index].positionZ;

	color = m_ModelInfoList[index].color;

	return;
}
```

Graphicsclass.h

[Same as previous codes]

The GraphicsClass for this tutorial includes a number of class we have used in the previous tutorials. It also includes the frustumclass.h and modellistclass.h header which are new.

```
#include "modellistclass.h"
#include "frustumclass.h"

class GraphicsClass
{
```

[Same as previous codes]

```
private:
```

[Same as previous codes]

Two of the new private class objects are the m_Frustum and m_ModelList.

```
	ModelListClass* m_ModelList;
	FrustumClass* m_Frustum;
};
```

Graphicsclass.cpp

[Same as previous codes]

The class constructor initializes the private member variables to null.

```
GraphicsClass::GraphicsClass()
{
```

[Same as previous codes]

```
	m_ModelList = 0;
	m_Frustum = 0;
}

bool GraphicsClass::Initialize(int screenWidth, int screenHeight, HWND hwnd)
{
```

[Same as previous codes]

```
	// Create the model object.
	m_Model = new ModelClass;
	if(!m_Model)
	{
		return false;
	}
```

We load a sphere model instead of a cube model for this tutorial.

```
	// Initialize the model object.
	result = m_Model->Initialize(m_D3D->GetDevice(), L"../Engine/data/seafloor.dds",
		"../Engine/data/sphere.txt");
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the model object.", L"Error", MB_OK);
		return false;
	}
```

[Same as previous codes]

Here we create the new ModelListClass object and have it create 25 randomly placed/colored sphere models.

```
	// Create the model list object.
	m_ModelList = new ModelListClass;
	if(!m_ModelList)
	{
		return false;
	}

	// Initialize the model list object.
	result = m_ModelList->Initialize(25);
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the model list object.", L"Error", MB_OK);
		return false;
	}
```

Here we create the new FrustumClass object. It doesn't need any initialization since that is done every frame using the ConstructFrustum function.

```
	// Create the frustum object.
	m_Frustum = new FrustumClass;
	if(!m_Frustum)
	{
		return false;
	}

	return true;
}
```

```
void GraphicsClass::Shutdown()
{
```

We release the new FrustumClass and ModelListClass objects here in the Shutdown function.

```
	// Release the frustum object.
	if(m_Frustum)
	{
		delete m_Frustum;
		m_Frustum = 0;
	}

	// Release the model list object.
	if(m_ModelList)
	{
		m_ModelList->Shutdown();
		delete m_ModelList;
		m_ModelList = 0;
	}

	// Release the light object.
	if(m_Light)
	{
		delete m_Light;
		m_Light = 0;
	}
```

[Same as previous codes]

```
}
```

The Frame function now takes in the rotation of the camera from the SystemClass that calls it. The position and rotation of the camera are then set so the view matrix can be properly updated in the Render function.

```
bool GraphicsClass::Frame(float rotationY)
{
	// Set the position of the camera.
	m_Camera->SetPosition(0.0f, 0.0f, -10.0f);

	// Set the rotation of the camera.
	m_Camera->SetRotation(0.0f, rotationY, 0.0f);

	return true;
}


bool GraphicsClass::Render()
{
	D3DXMATRIX worldMatrix, viewMatrix, projectionMatrix, orthoMatrix;
	int modelCount, renderCount, index;
	float positionX, positionY, positionZ, radius;
	D3DXVECTOR4 color;
	bool renderModel, result;

	// Clear the buffers to begin the scene.
	m_D3D->BeginScene(0.0f, 0.0f, 0.0f, 1.0f);

	// Generate the view matrix based on the camera's position.
	m_Camera->Render();

	// Get the world, view, projection, and ortho matrices from the camera and d3d objects.
	m_D3D->GetWorldMatrix(worldMatrix);
	m_Camera->GetViewMatrix(viewMatrix);
	m_D3D->GetProjectionMatrix(projectionMatrix);
	m_D3D->GetOrthoMatrix(orthoMatrix);
```

The major change to the Render function is that we now construct the viewing frustum each frame based on the updated viewing matrix. This construction has to occur each time the view matrix changes or the frustum culling checks we do will not be correct.

```
// Construct the frustum.
m_Frustum->ConstructFrustum(SCREEN_DEPTH, projectionMatrix, viewMatrix);

// Get the number of models that will be rendered.
modelCount = m_ModelList->GetModelCount();

// Initialize the count of models that have been rendered.
renderCount = 0;
```

Now loop through all the models in the ModelListClass object.

```
// Go through all the models and render them only if they can be seen by the camera view.
for(index=0; index<modelCount; index++)
{
	// Get the position and color of the sphere model at this index.
	m_ModelList->GetData(index, positionX, positionY, positionZ, color);

	// Set the radius of the sphere to 1.0 since this is already known.
	radius = 1.0f;
```

Here is where we use the new FrustumClass object. We check if the sphere is viewable in the viewing frustum. If it can be seen we render it, if it cannot be seen we skip it and check the next one. This is where we will gain all the speed by using frustum culling.

```
	// Check if the sphere model is in the view frustum.
	renderModel = m_Frustum->CheckSphere(positionX, positionY, positionZ, radius);

	// If it can be seen then render it, if not skip this model and check the next sphere.
	if(renderModel)
	{
		// Move the model to the location it should be rendered at.
		D3DXMatrixTranslation(&worldMatrix, positionX, positionY, positionZ);

		// Put the model vertex and index buffers on the graphics pipeline to prepare them for drawing.
		m_Model->Render(m_D3D->GetDeviceContext());

		// Render the model using the light shader.
		m_LightShader->Render(m_D3D->GetDeviceContext(), m_Model->GetIndexCount(),
		worldMatrix, viewMatrix, projectionMatrix, m_Model->GetTexture(), m_Light-
		>GetDirection(), color);

		// Reset to the original world matrix.
		m_D3D->GetWorldMatrix(worldMatrix);

		// Since this model was rendered then increase the count for this frame.
		renderCount++;
	}
}
```

We use the slightly modified TextClass to display how many spheres were actually rendered. We can also infer for this number that the spheres that were not rendered were instead culled using the new FrustumClass object.

```
// Set the number of models that was actually rendered this frame.
result = m_Text->SetRenderCount(renderCount, m_D3D->GetDeviceContext());
if(!result)
{
	return false;
}

// Turn off the Z buffer to begin all 2D rendering.
m_D3D->TurnZBufferOff();
```

```
	// Turn on the alpha blending before rendering the text.
	m_D3D->TurnOnAlphaBlending();

	// Render the text string of the render count.
	m_Text->Render(m_D3D->GetDeviceContext(), worldMatrix, orthoMatrix);
	if(!result)
	{
		return false;
	}

	// Turn off alpha blending after rendering the text.
	m_D3D->TurnOffAlphaBlending();

	// Turn the Z buffer back on now that all 2D rendering has completed.
	m_D3D->TurnZBufferOn();

	// Present the rendered scene to the screen.
	m_D3D->EndScene();

	return true;
}
```

Positionclass.h

To allow for camera movement by using the left and right arrow key in this tutorial we create a new class to calculate and maintain the position of the viewer. This class will only handle turning left and right for now but can be expanded to maintain all different movement changes. The movement also includes acceleration and deceleration to create a smooth camera effect.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: positionclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _POSITIONCLASS_H_
#define _POSITIONCLASS_H_


//////////////
// INCLUDES //
//////////////
#include <math.h>


////////////////////////////////////////////////////////////////////////////////
// Class name: PositionClass
////////////////////////////////////////////////////////////////////////////////
class PositionClass
{
public:
	PositionClass();
	PositionClass(const PositionClass&);
	~PositionClass();

	void SetFrameTime(float);
	void GetRotation(float&);

	void TurnLeft(bool);
	void TurnRight(bool);

private:
	float m_frameTime;
	float m_rotationY;
	float m_leftTurnSpeed, m_rightTurnSpeed;
};

#endif
```

Positionclass.cpp

```
////////////////////////////////////////////////////////////////////////////////
// Filename: positionclass.cpp
////////////////////////////////////////////////////////////////////////////////
#include "positionclass.h"
```

The class constructor initializes the private member variables to zero to start with.

```
PositionClass::PositionClass()
{
	m_frameTime = 0.0f;
	m_rotationY = 0.0f;
	m_leftTurnSpeed    = 0.0f;
	m_rightTurnSpeed = 0.0f;
}

PositionClass::PositionClass(const PositionClass& other)
{
}


PositionClass::~PositionClass()
{
}
```

The SetFrameTime function is used to set the frame speed in this class. PositionClass will use that frame time speed to calculate how fast the viewer should be moving and rotating. This function should always be called at the beginning of each frame before using this class to move the viewing position.

```
void PositionClass::SetFrameTime(float time)
{
	m_frameTime = time;
	return;
}
```

GetRotation returns the Y-axis rotation of the viewer. This is the only helper function we need for this tutorial but could be expanded to get more information about the location of the viewer.

```
void PositionClass::GetRotation(float& y)
{
	y = m_rotationY;
	return;
}
```

The movement functions both work the same. Both functions are called each frame. The keydown input variable to each function indicates if the user is pressing the left key or the right key. If they are pressing the key then each frame the speed will accelerate until it hits a maximum. This way the camera speeds up similar to the acceleration in a vehicle creating the effect of smooth movement and high responsiveness. Likewise if the user releases the key and the keydown variable is false it will then smoothly slow down each frame until the speed hits zero. The speed is calculated against the frame time to ensure the movement speed remains the same regardless of the frame rate. Each function then uses some basic math to calculate the new position of the camera.

```
void PositionClass::TurnLeft(bool keydown)
{
	// If the key is pressed increase the speed at which the camera turns left.    If not slow down the turn speed.
	if(keydown)
	{
		m_leftTurnSpeed += m_frameTime * 0.01f;

		if(m_leftTurnSpeed > (m_frameTime * 0.15f))
		{
			m_leftTurnSpeed = m_frameTime * 0.15f;
		}
```

```
	}
	else
	{
		m_leftTurnSpeed -= m_frameTime* 0.005f;

		if(m_leftTurnSpeed < 0.0f)
		{
			m_leftTurnSpeed = 0.0f;
		}
	}

	// Update the rotation using the turning speed.
	m_rotationY -= m_leftTurnSpeed;
	if(m_rotationY < 0.0f)
	{
		m_rotationY += 360.0f;
	}

	return;
}

void PositionClass::TurnRight(bool keydown)
{
	// If the key is pressed increase the speed at which the camera turns right. If not slow down the turn speed.
	if(keydown)
	{
		m_rightTurnSpeed += m_frameTime * 0.01f;

		if(m_rightTurnSpeed > (m_frameTime * 0.15f))
		{
			m_rightTurnSpeed = m_frameTime * 0.15f;
		}
	}
	else
	{
		m_rightTurnSpeed -= m_frameTime* 0.005f;

		if(m_rightTurnSpeed < 0.0f)
		{
			m_rightTurnSpeed = 0.0f;
		}
	}

	// Update the rotation using the turning speed.
	m_rotationY += m_rightTurnSpeed;
	if(m_rotationY > 360.0f)
	{
		m_rotationY -= 360.0f;
	}

	return;
}
```

Systemclass.h

The SystemClass has been modified to use the new PostionClass.

[Same as previous codes]

```
#include "positionclass.h"

class SystemClass
{
```

[Same as previous codes]

```
private:
```

[Same as previous codes]

```
	TimerClass* m_Timer;
	PositionClass* m_Position;
};
```

Systemclass.cpp

```
SystemClass::SystemClass()
{
```

[Same as previous codes]

The new PositionClass object is initialized to null in the class constructor.

```
	m_Position = 0;
}

bool SystemClass::Initialize()
{
```

[Same as previous codes]

```
	// Initialize the timer object.
	result = m_Timer->Initialize();
	if(!result)
	{
		MessageBox(m_hwnd, L"Could not initialize the Timer object.", L"Error", MB_OK);
		return false;
	}
```

Create the new PositionClass object here. It doesn't require any initialization.

```
	// Create the position object.
	m_Position = new PositionClass;
	if(!m_Position)
	{
		return false;
	}

	return true;
}

void SystemClass::Shutdown()
{
```

The PositionClass object is released here in the Shutdown function.

```
	// Release the position object.
	if(m_Position)
	{
		delete m_Position;
		m_Position = 0;
	}
```

[Same as previous codes]

```
}

bool SystemClass::Frame()
{
```

```
	bool keyDown, result;
	float rotationY;

	// Update the system stats.
	m_Timer->Frame();

	// Do the input frame processing.
	result = m_Input->Frame();
	if(!result)
	{
		return false;
	}
```

During each frame the PositionClass object is update with the frame time.

```
	// Set the frame time for calculating the updated position.
	m_Position->SetFrameTime(m_Timer->GetTime());
```

After the frame time update the PositionClass movement functions can be updated with the current state of the keyboard. The movement functions will update the position of the camera to the new location for this frame.

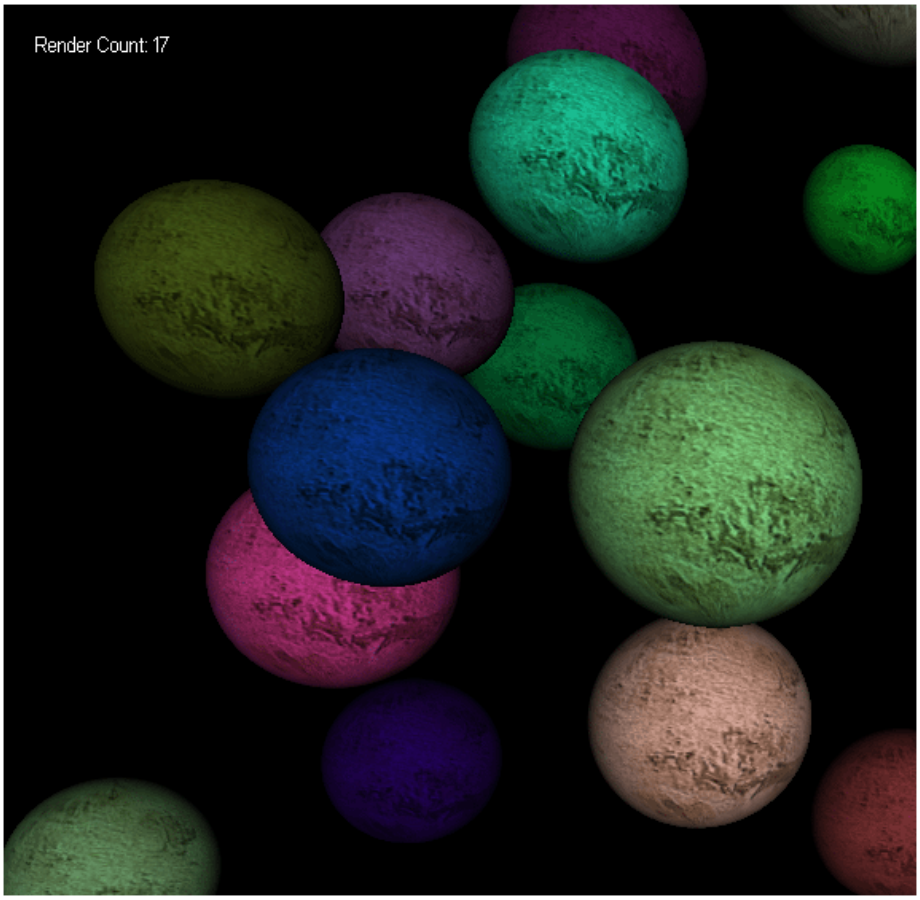
```
	// Check if the left or right arrow key has been pressed, if so rotate the camera accordingly.
	keyDown = m_Input->IsLeftArrowPressed();
	m_Position->TurnLeft(keyDown);

	keyDown = m_Input->IsRightArrowPressed();
	m_Position->TurnRight(keyDown);
```

The new rotation of the camera is retrieved and sent to the Graphics::Frame function to update the camera position.

```
	// Get the current view point rotation.
	m_Position->GetRotation(rotationY);

	// Do the frame processing for the graphics object.
	result = m_Graphics->Frame(rotationY);
	if(!result)
	{
		return false;
	}

	// Finally render the graphics to the screen.
	result = m_Graphics->Render();
	if(!result)
	{
		return false;
	}

	return true;
}
```

Summary

Now you have seen how to cull objects. The only trick from here is determining whether a cube, rectangle, sphere, or clever use of a point is better for culling your different objects.



To Do Exercises

1. Recompile and run the program. Use the left and right arrow key to move the camera and update the render count in the upper left corner.

2. Load the cube model instead and change the cull check to CheckCube.

3. Create some different models and test which of the culling checks works best for them.

[Original script: www.rastertek.com]