# Tutorial: Picking

In most 3D applications the user will need to click on the screen with the mouse to select or interact with one of the 3D objects in the scene. This process is usually referred to as selection or picking. This tutorial will cover how to implement picking using DirectX 11.

The process of picking involves translating a 2D mouse coordinate position into a vector that is in world space. That vector is then used for intersection checks with all the visible 3D objects. Once the 3D object is determined the test can be further refined to determine exactly which polygon was selected on that 3D object.

For this tutorial we will use a single sphere and do a ray-sphere intersection test whenever the user presses the left mouse button.

## Applicationclass.h

```
////////////////////////////////////////////////////////////////////////////////
// Filename: applicationclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _APPLICATIONCLASS_H_
#define _APPLICATIONCLASS_H_


//////////////////////
// MY CLASS INCLUDES //
//////////////////////
#include "inputclass.h"
#include "d3dclass.h"
#include "cameraclass.h"
#include "modelclass.h"
#include "textureshaderclass.h"
#include "lightshaderclass.h"
#include "lightclass.h"
#include "textclass.h"
#include "bitmapclass.h"


/////////////
// GLOBALS //
/////////////
const bool FULL_SCREEN = true;
const bool VSYNC_ENABLED = true;
const float SCREEN_DEPTH = 1000.0f;
const float SCREEN_NEAR = 0.1f;


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

	bool Initialize(HINSTANCE, HWND, int, int);
	void Shutdown();
	bool Frame();

private:
	bool HandleInput();
	bool Render();
```

We have two new functions here. The first one is the general intersection check that forms the vector for checking the intersection and then calls the specific type of intersection check required. The second function is the ray-sphere

intersection check function; this function is called by TestIntersection. For other intersection tests such as ray-triangle, ray-rectangle, and so forth you would add them here.

```
        void TestIntersection(int, int);
        bool RaySphereIntersect(D3DXVECTOR3, D3DXVECTOR3, float);

private:
        InputClass* m_Input;
        D3DClass* m_D3D;
        CameraClass* m_Camera;
```

The model we will be loading into this object will be a sphere.

```
        ModelClass* m_Model;
        TextureShaderClass* m_TextureShader;
        LightShaderClass* m_LightShader;
        LightClass* m_Light;
        TextClass* m_Text;
```

The bitmap will be used to draw a mouse cursor so that the user knows where they are clicking on the screen.

```
        BitmapClass* m_Bitmap;
```

The m_beginCheck variable is used to determine if the user has clicked on the screen or not.

```
        bool m_beginCheck;
        int m_screenWidth, m_screenHeight;
};

#endif
```

## Applicationclass.cpp

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


ApplicationClass::ApplicationClass()
{
        m_Input = 0;
        m_D3D = 0;
        m_Camera = 0;
        m_Model = 0;
        m_TextureShader = 0;
        m_LightShader = 0;
        m_Light = 0;
        m_Text = 0;
        m_Bitmap = 0;
}


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


ApplicationClass::~ApplicationClass()
{
}
```

```
bool ApplicationClass::Initialize(HINSTANCE hinstance, HWND hwnd, int screenWidth, int screenHeight)
{
	bool result;
	D3DXMATRIX baseViewMatrix;


	// Store the screen width and height.
	m_screenWidth = screenWidth;
	m_screenHeight = screenHeight;

	// Create the input object.
	m_Input = new InputClass;
	if(!m_Input)
	{
		return false;
	}

	// Initialize the input object.
	result = m_Input->Initialize(hinstance, hwnd, screenWidth, screenHeight);
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the input object.", L"Error", MB_OK);
		return false;
	}

	// Create the Direct3D object.
	m_D3D = new D3DClass;
	if(!m_D3D)
	{
		return false;
	}

	// Initialize the Direct3D object.
	result = m_D3D->Initialize(screenWidth, screenHeight, VSYNC_ENABLED, hwnd, FULL_SCREEN,
SCREEN_DEPTH, SCREEN_NEAR);
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize Direct3D.", L"Error", MB_OK);
		return false;
	}

	// Create the camera object.
	m_Camera = new CameraClass;
	if(!m_Camera)
	{
		return false;
	}

	// Set the initial position of the camera.
	m_Camera->SetPosition(0.0f, 0.0f, -10.0f);
	m_Camera->Render();
	m_Camera->GetViewMatrix(baseViewMatrix);
```

We load the sphere model here.

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

	// Initialize the model object.
	result = m_Model->Initialize(m_D3D->GetDevice(), "../Engine/data/sphere.txt", L"../Engine/data/blue.dds");
```

```
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the model object.", L"Error", MB_OK);
		return false;
	}

	// Create the texture shader object.
	m_TextureShader = new TextureShaderClass;
	if(!m_TextureShader)
	{
		return false;
	}

	// Initialize the texture shader object.
	result = m_TextureShader->Initialize(m_D3D->GetDevice(), hwnd);
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the texture shader object.", L"Error", MB_OK);
		return false;
	}

	// Create the light shader object.
	m_LightShader = new LightShaderClass;
	if(!m_LightShader)
	{
		return false;
	}

	// Initialize the light shader object.
	result = m_LightShader->Initialize(m_D3D->GetDevice(), hwnd);
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the light shader object.", L"Error", MB_OK);
		return false;
	}

	// Create the light object.
	m_Light = new LightClass;
	if(!m_Light)
	{
		return false;
	}

	// Initialize the light object.
	m_Light->SetDirection(0.0f, 0.0f, 1.0f);

	// Create the text object.
	m_Text = new TextClass;
	if(!m_Text)
	{
		return false;
	}

	// Initialize the text object.
	result = m_Text->Initialize(m_D3D->GetDevice(), m_D3D->GetDeviceContext(), hwnd, screenWidth,
screenHeight, baseViewMatrix);
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the text object.", L"Error", MB_OK);
		return false;
	}
```

We load the mouse cursor here.

```
	// Create the bitmap object.
	m_Bitmap = new BitmapClass;
```

```
	if(!m_Bitmap)
	{
		return false;
	}

	// Initialize the bitmap object.
	result = m_Bitmap->Initialize(m_D3D->GetDevice(), screenWidth, screenHeight,
L"../Engine/data/mouse.dds", 32, 32);
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the bitmap object.", L"Error", MB_OK);
		return false;
	}

	// Initialize that the user has not clicked on the screen to try an intersection test yet.
	m_beginCheck = false;

	return true;
}


void ApplicationClass::Shutdown()
{
	// Release the bitmap object.
	if(m_Bitmap)
	{
		m_Bitmap->Shutdown();
		delete m_Bitmap;
		m_Bitmap = 0;
	}

	// Release the text object.
	if(m_Text)
	{
		m_Text->Shutdown();
		delete m_Text;
		m_Text = 0;
	}

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

	// Release the light shader object.
	if(m_LightShader)
	{
		m_LightShader->Shutdown();
		delete m_LightShader;
		m_LightShader = 0;
	}

	// Release the texture shader object.
	if(m_TextureShader)
	{
		m_TextureShader->Shutdown();
		delete m_TextureShader;
		m_TextureShader = 0;
	}

	// Release the model object.
	if(m_Model)
	{
		m_Model->Shutdown();
```

```
		delete m_Model;
		m_Model = 0;
	}

	// Release the camera object.
	if(m_Camera)
	{
		delete m_Camera;
		m_Camera = 0;
	}

	// Release the D3D object.
	if(m_D3D)
	{
		m_D3D->Shutdown();
		delete m_D3D;
		m_D3D = 0;
	}

	// Release the input object.
	if(m_Input)
	{
		m_Input->Shutdown();
		delete m_Input;
		m_Input = 0;
	}

	return;
}


bool ApplicationClass::Frame()
{
	bool result;


	// Handle the input processing.
	result = HandleInput();
	if(!result)
	{
		return false;
	}

	// Render the graphics scene.
	result = Render();
	if(!result)
	{
		return false;
	}

	return true;
}


bool ApplicationClass::HandleInput()
{
	bool result;
	int mouseX, mouseY;


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

```
	// Check if the user pressed escape and wants to exit the application.
	if(m_Input->IsEscapePressed() == true)
	{
		return false;
	}
```

In the input handling that is called each frame we now check to see if the user has pressed or released the left mouse button. If they have pressed the left mouse button then we perform the intersection check with the sphere using the current 2D mouse coordinates.

```
	// Check if the left mouse button has been pressed.
	if(m_Input->IsLeftMouseButtonDown() == true)
	{
		// If they have clicked on the screen with the mouse then perform an intersection test.
		if(m_beginCheck == false)
		{
			m_beginCheck = true;
			m_Input->GetMouseLocation(mouseX, mouseY);
			TestIntersection(mouseX, mouseY);
		}
	}

	// Check if the left mouse button has been released.
	if(m_Input->IsLeftMouseButtonDown() == false)
	{
		m_beginCheck = false;
	}

	return true;
}


bool ApplicationClass::Render()
{
	D3DXMATRIX worldMatrix, viewMatrix, projectionMatrix, orthoMatrix, translateMatrix;
	bool result;
	int mouseX, mouseY;


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

We translate to the position of the sphere and then render it.

```
	// Translate to the location of the sphere.
	D3DXMatrixTranslation(&translateMatrix, -5.0f, 1.0f, 5.0f);
	D3DXMatrixMultiply(&worldMatrix, &worldMatrix, &translateMatrix);

	// Render the model using the light shader.
	m_Model->Render(m_D3D->GetDeviceContext());
	result = m_LightShader->Render(m_D3D->GetDeviceContext(), m_Model->GetIndexCount(), worldMatrix,
viewMatrix, projectionMatrix, m_Model->GetTexture(), m_Light->GetDirection());
	if(!result)
	{
		return false;
	}
```

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

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

	// Turn on alpha blending.
	m_D3D->EnableAlphaBlending();
```

Here we get the position of the mouse and then render the bitmap of the mouse cursor using the 2D coordinates from the m_Input object.

```
	// Get the location of the mouse from the input object,
	m_Input->GetMouseLocation(mouseX, mouseY);

	// Render the mouse cursor with the texture shader.
	result = m_Bitmap->Render(m_D3D->GetDeviceContext(), mouseX, mouseY);    if(!result) { return false; }
	result = m_TextureShader->Render(m_D3D->GetDeviceContext(), m_Bitmap->GetIndexCount(),
worldMatrix, viewMatrix, orthoMatrix, m_Bitmap->GetTexture());

	// Render the text strings.
	result = m_Text->Render(m_D3D->GetDeviceContext(), worldMatrix, orthoMatrix);
	if(!result)
	{
		return false;
	}

	// Turn of alpha blending.
	m_D3D->DisableAlphaBlending();

	// 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;
}
```

The TestIntersection function is pretty much the entire focus of this tutorial. It takes as input the 2D mouse coordinates and then forms a vector in 3D space which it uses to then check for an intersection with the sphere. That vector is called the picking ray. The picking ray has an origin and a direction. With the 3D coordinate (origin) and 3D vector/normal (direction) we can create a line in 3D space and find out what it collides with.

In the other HLSL tutorials we are very used to a vertex shader that takes a 3D point (vertice) and moves it from 3D space onto the 2D screen so it can be rendered as a pixel. Well now we are doing the exact opposite and moving a 2D point from the screen into 3D space. So what we need to do is just reverse our usual process. So where we would usually take a 3D point from world to view to projection to make a 2D point, we will now instead take a 2D point and go from projection to view to world and turn it into a 3D point.

To do the reverse process we first start by taking the mouse coordinates and moving them into the -1 to +1 range on both axis. When we have that we then divide by the screen aspect using the projection matrix. With that value we can then multiply it by the inverse view matrix (inverse because we are going in reverse direction) to get the direction vector in view space. We can set the origin of the vector in view space to just be the location of the camera.

With the direction vector and origin in view space we can now complete the final process of moving it into 3D world space. To do so we first need to get the world matrix and translate it by the position of the sphere. With the updated world matrix we once again need to invert it (since the process is going in the opposite direction) and then we can multiply the origin and direction by the inverted world matrix. We also normalize the direction after the multiplication. This gives us the origin and direction of the vector in 3D world space so that we can do tests with other objects that are also in 3D world space.

Now that we have the origin of the vector and the direction of the vector we can perform an intersection test. In this tutorial we perform a ray-sphere intersection test, but you could perform any kind of intersection test now that you have the vector in 3D world space.

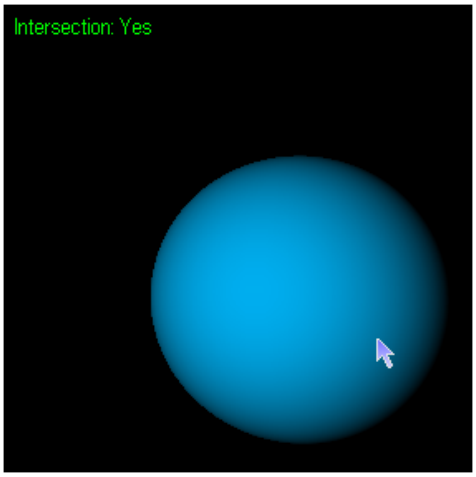
```
void ApplicationClass::TestIntersection(int mouseX, int mouseY)
{
	float pointX, pointY;
	D3DXMATRIX projectionMatrix, viewMatrix, inverseViewMatrix, worldMatrix, translateMatrix,
inverseWorldMatrix;
	D3DXVECTOR3 direction, origin, rayOrigin, rayDirection;
	bool intersect, result;


	// Move the mouse cursor coordinates into the -1 to +1 range.
	pointX = ((2.0f * (float)mouseX) / (float)m_screenWidth) - 1.0f;
	pointY = (((2.0f * (float)mouseY) / (float)m_screenHeight) - 1.0f) * -1.0f;

	// Adjust the points using the projection matrix to account for the aspect ratio of the viewport.
	m_D3D->GetProjectionMatrix(projectionMatrix);
	pointX = pointX / projectionMatrix._11;
	pointY = pointY / projectionMatrix._22;

	// Get the inverse of the view matrix.
	m_Camera->GetViewMatrix(viewMatrix);
	D3DXMatrixInverse(&inverseViewMatrix, NULL, &viewMatrix);

	// Calculate the direction of the picking ray in view space.
	direction.x = (pointX * inverseViewMatrix._11) + (pointY * inverseViewMatrix._21) + inverseViewMatrix._31;
	direction.y = (pointX * inverseViewMatrix._12) + (pointY * inverseViewMatrix._22) + inverseViewMatrix._32;
	direction.z = (pointX * inverseViewMatrix._13) + (pointY * inverseViewMatrix._23) + inverseViewMatrix._33;

	// Get the origin of the picking ray which is the position of the camera.
	origin = m_Camera->GetPosition();

	// Get the world matrix and translate to the location of the sphere.
	m_D3D->GetWorldMatrix(worldMatrix);
	D3DXMatrixTranslation(&translateMatrix, -5.0f, 1.0f, 5.0f);
	D3DXMatrixMultiply(&worldMatrix, &worldMatrix, &translateMatrix);

	// Now get the inverse of the translated world matrix.
	D3DXMatrixInverse(&inverseWorldMatrix, NULL, &worldMatrix);

	// Now transform the ray origin and the ray direction from view space to world space.
	D3DXVec3TransformCoord(&rayOrigin, &origin, &inverseWorldMatrix);
	D3DXVec3TransformNormal(&rayDirection, &direction, &inverseWorldMatrix);

	// Normalize the ray direction.
	D3DXVec3Normalize(&rayDirection, &rayDirection);

	// Now perform the ray-sphere intersection test.
	intersect = RaySphereIntersect(rayOrigin, rayDirection, 1.0f);

	if(intersect == true)
	{
		// If it does intersect then set the intersection to "yes" in the text string that is displayed to the
screen.
		result = m_Text->SetIntersection(true, m_D3D->GetDeviceContext());
	}
	else
	{
		// If not then set the intersection to "No".
		result = m_Text->SetIntersection(false, m_D3D->GetDeviceContext());
	}

	return;
}
```

This function performs the math of a basic ray-sphere intersection test.

```
bool ApplicationClass::RaySphereIntersect(D3DXVECTOR3 rayOrigin, D3DXVECTOR3 rayDirection, float radius)
{
	float a, b, c, discriminant;


	// Calculate the a, b, and c coefficients.
	a = (rayDirection.x * rayDirection.x) + (rayDirection.y * rayDirection.y) + (rayDirection.z * rayDirection.z);
	b = ((rayDirection.x * rayOrigin.x) + (rayDirection.y * rayOrigin.y) + (rayDirection.z * rayOrigin.z)) * 2.0f;
	c = ((rayOrigin.x * rayOrigin.x) + (rayOrigin.y * rayOrigin.y) + (rayOrigin.z * rayOrigin.z)) - (radius * radius);

	// Find the discriminant.
	discriminant = (b * b) - (4 * a * c);

	// if discriminant is negative the picking ray missed the sphere, otherwise it intersected the sphere.
	if (discriminant < 0.0f)
	{
		return false;
	}

	return true;
}
```

## Summary

We can now perform basic intersection tests with 3D objects in the scene using picking.




## To Do Exercises

1. Compile and run the program. Use the mouse to move the cursor and left click on the sphere or the empty space to test intersections. Press escape to quit.

2. Add a cube, triangle, and rectangle model to the scene. Also add intersection test functions for the new three types.

3. Now that you have "bounding box" tests further refine it so that if the line intersects the sphere it then does a second check for all the triangles in the sphere and highlights the selected triangle.

4. Do the same as number three expect for rectangles and cubes.

5. Place two objects in front of each other, make sure you intersection test returns only the one closest to the camera and ignores the objects behind it if the intersection test returns multiple intersections.

*[Original script: www.rastertek.com]*