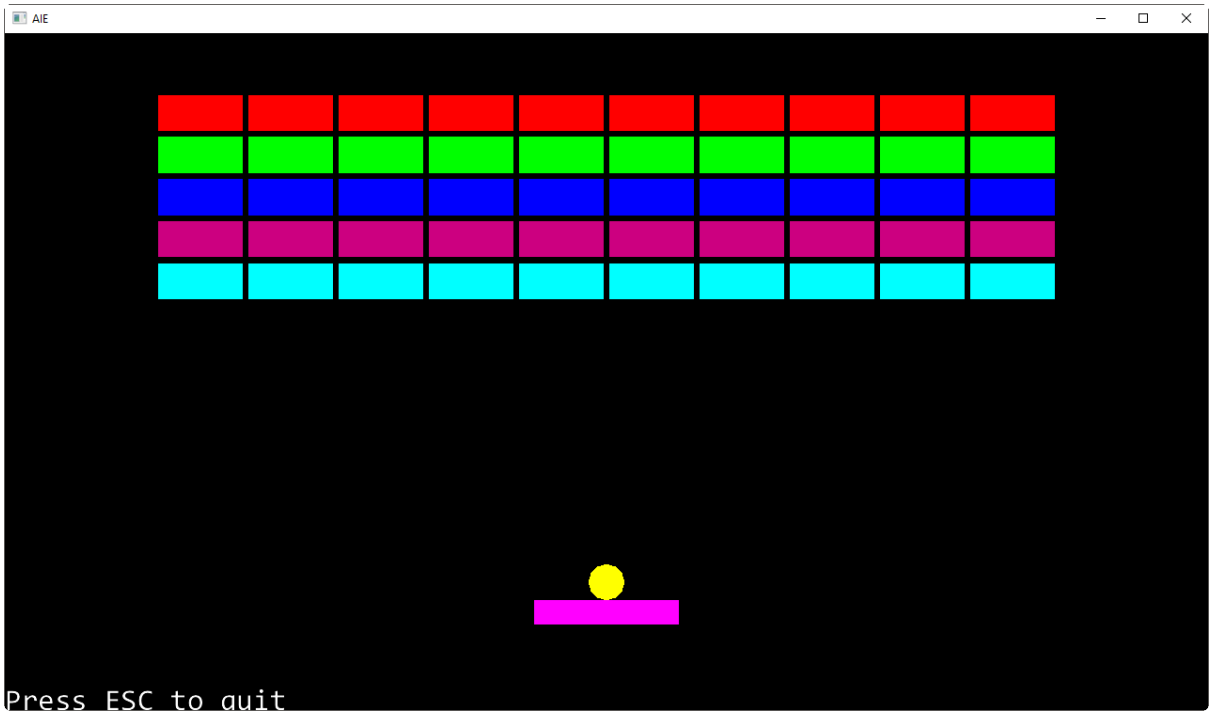
Tutorial – Introduction to Physics

Introduction:

In this tutorial we are going to be familiarizing ourselves with the 2D drawing functionality of the AIE Bootstrap framework. Hopefully you will already be familiar with this framework, so for the most part this should be a bit of revision.

We won't be concerning ourselves with any physics in this lecture. Instead, we will focus on drawing some 2D shapes. We'll create a *Breakout* clone screen (not a game) by arranging some shapes in a 2D world.

While our program won't be interactive and will contain no game logic, you could extend this program throughout this course to create your own *Breakout* clone game.



Downloading and Installing:

If you haven't already done so, download the AIE Bootstrap framework from GitHub via this link:
https://github.com/AcademyOfInteractiveEntertainment/aieBootstrap/archive/master.zip
This will directly download the framework for you to begin working with.

Even if you already have a copy of this project, it is a good idea to download a fresh copy. This project is continually being updated, and the latest changes add support for Visual Studio 2017.

Alternatively, you may wish to set up your own Git project and clone the repository. Cloning the repository is not necessary, although it will allow you to be notified when updates and changes are made to the framework.

The landing page for this project contains some additional information, including video tutorials on how to set up a Git project: https://github.com/AcademyOfInteractiveEntertainment/aieBootstrap

Once you have download the project, extract it to a folder on your machine. The project comes complete with a solution and a 2D and 3D demonstration project.

Before opening the solution, install the project templates if you have not already done so.

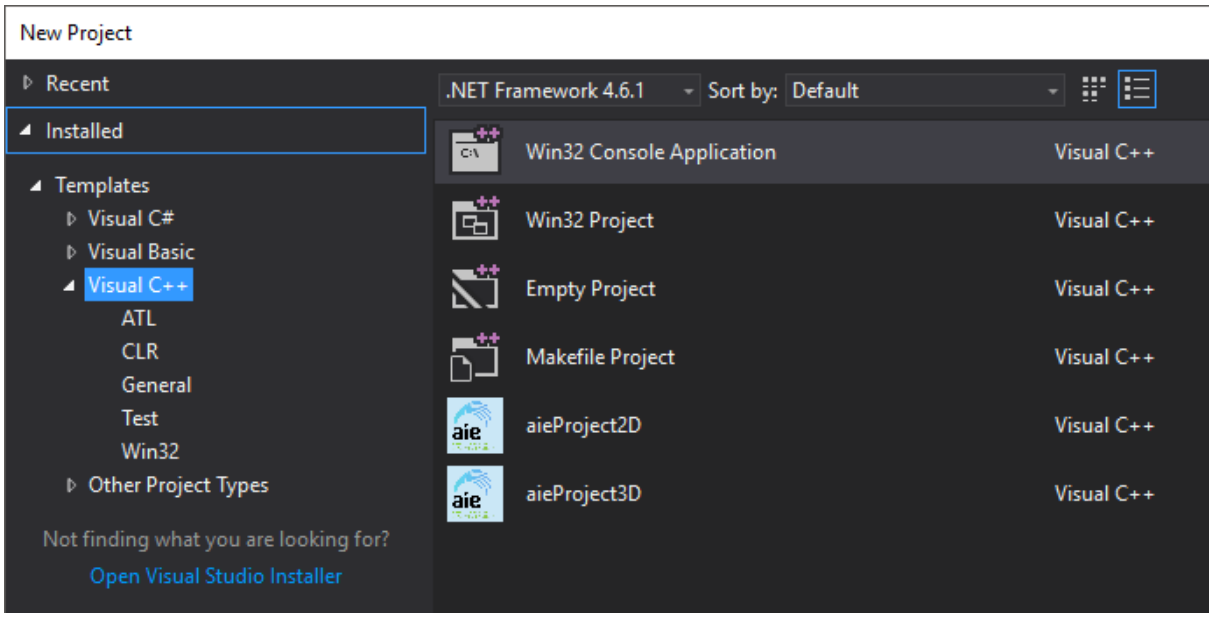
Locate the project templates in the *aieBootstrap-master\tools\ProjectTemplates* folder.

Copy the two *.zip* files into the Visual Studio templates directory. You'll find this in your *MyDocuments* folder, for example:

Documents\Visual Studio 2017\Templates\ProjectTemplates\Visual C++ Project\

Copy both of the *.zip* files to this location. Do not extract the *.zip* files.

After restarting Visual Studio you will now see two new project templates available whenever you create a new Visual C++ project.



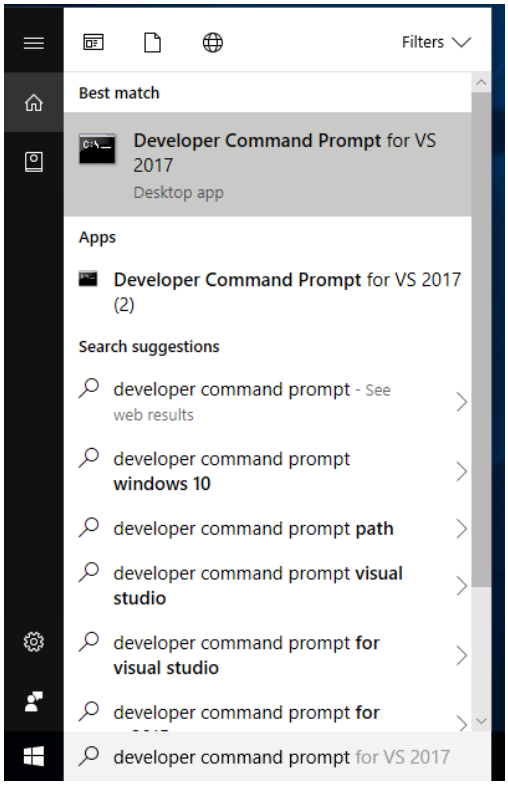
Clearing the Template Cache:

If you already have the templates installed and you wish to update them, you'll need to clear the Visual Studio template cache.

The easiest way to do this is to use the *Developer Command Prompt for Visual Studio 2017.*

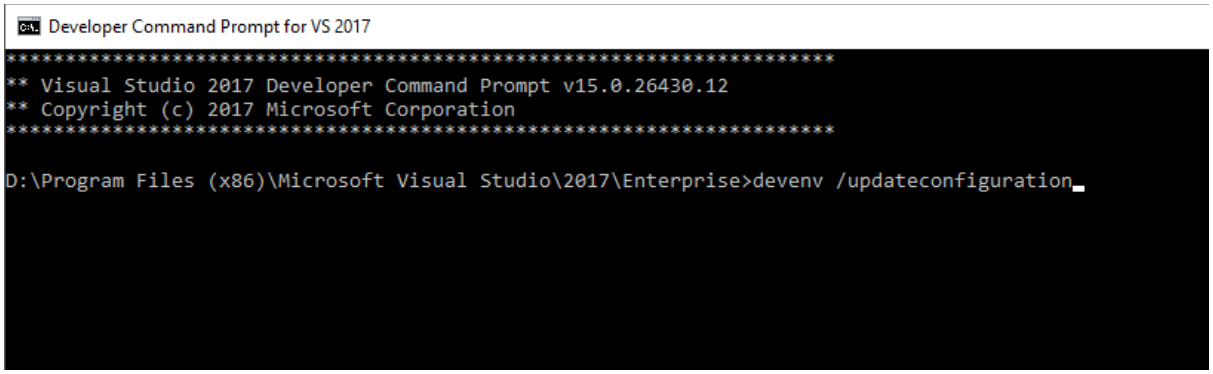
Before beginning, make sure Visual Studio is closed.

From the Windows *Start* menu, search for and open the *Developer Command Prompt for VS 2017*



This will launch a console window. Type the following command:

devenv /updateconfiguration

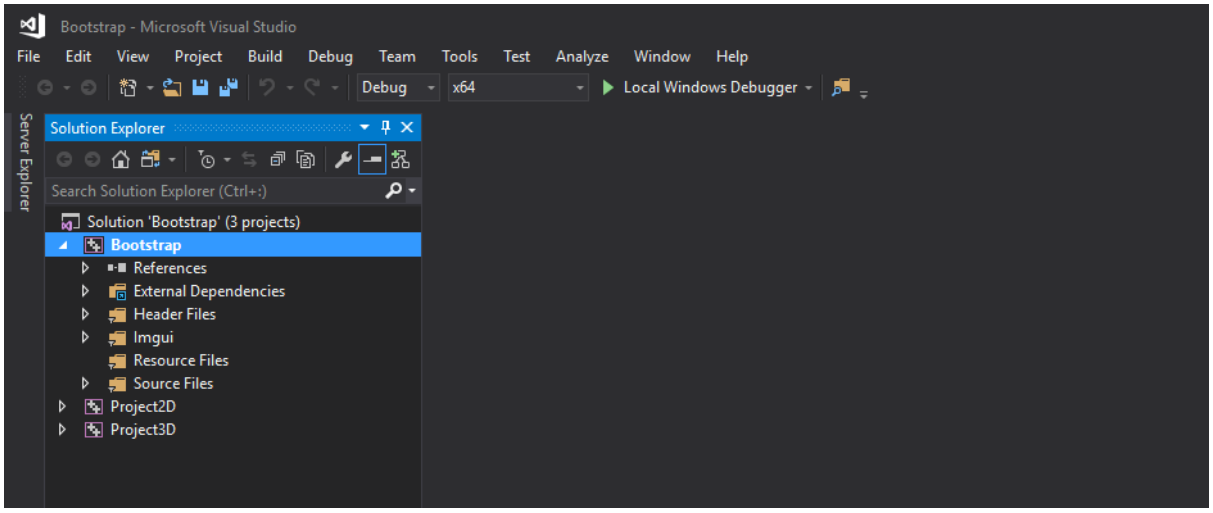


This will update the project template cache. Open Visual Studio again and you should now see both the 2D and 3D bootstrap project templates when you create a new project.

Working with AIE Bootstrap:

The AIE Bootstrap framework comes with a provided Visual Studio solution (Bootstrap.sln).

Open this solution in Visual Studio, or double-click on the .sln file:



The *Bootstrap* project is the framework itself. Every 2D or 3D project you make will link with this library to enable features like drawing using OpenGL, or handling keyboard input.

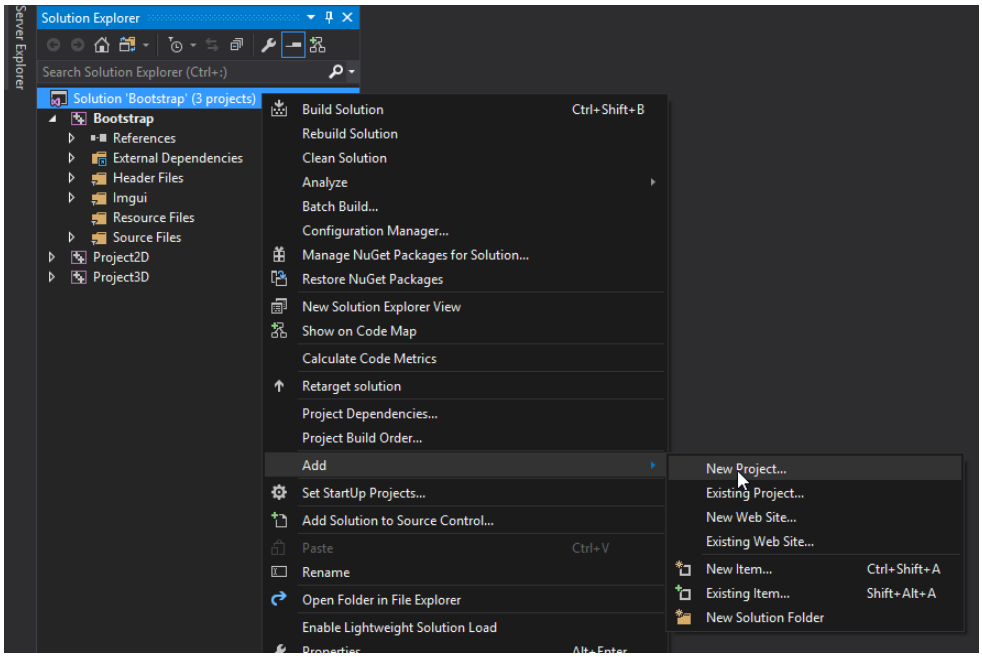
You will need to build the *Bootstrap* project. You can also build and execute the 2D and 3D projects if you wish to run these provided demonstration programs.

Creating a New Project:

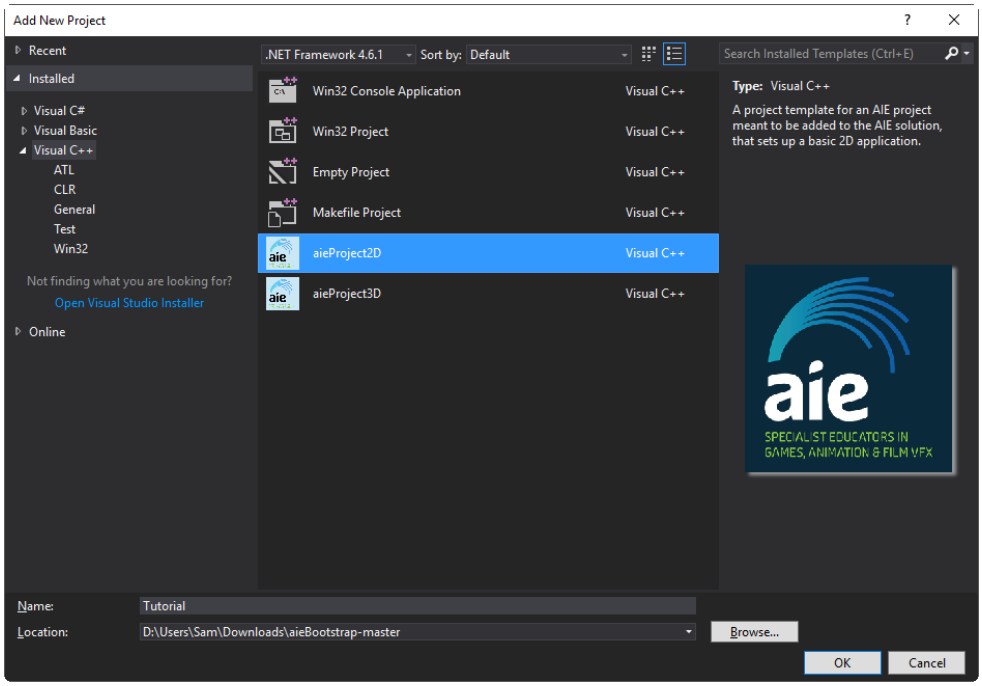
When working on a new tutorial, you will need to add your project to this solution.

Adding your project to this solution is much easier than creating a new solution for each tutorial and trying to import the *Bootstrap* library manually.

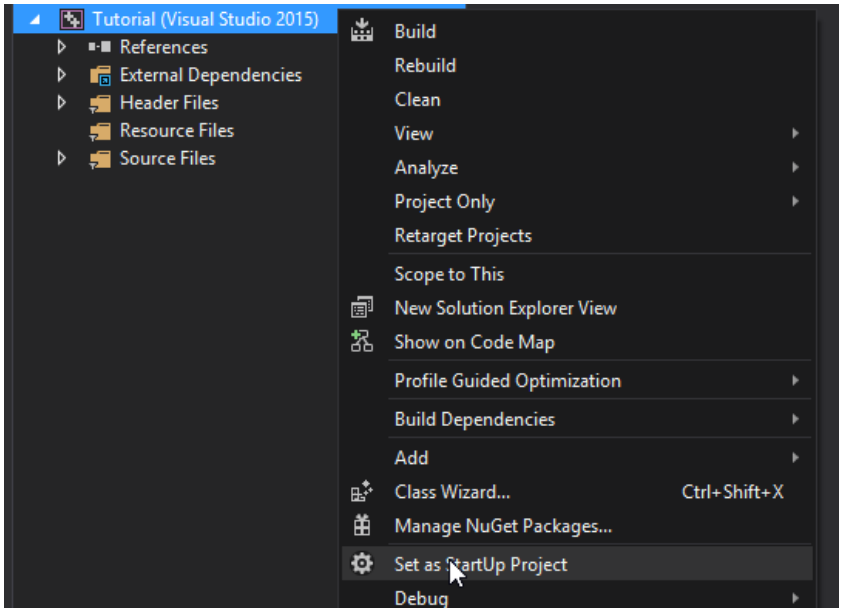
To create a new project, right-click on the solution name in the *Solution Explorer* and select *Add -> New Project*.



Select the 2D project template (since we will be creating our physics simulation in 2D):



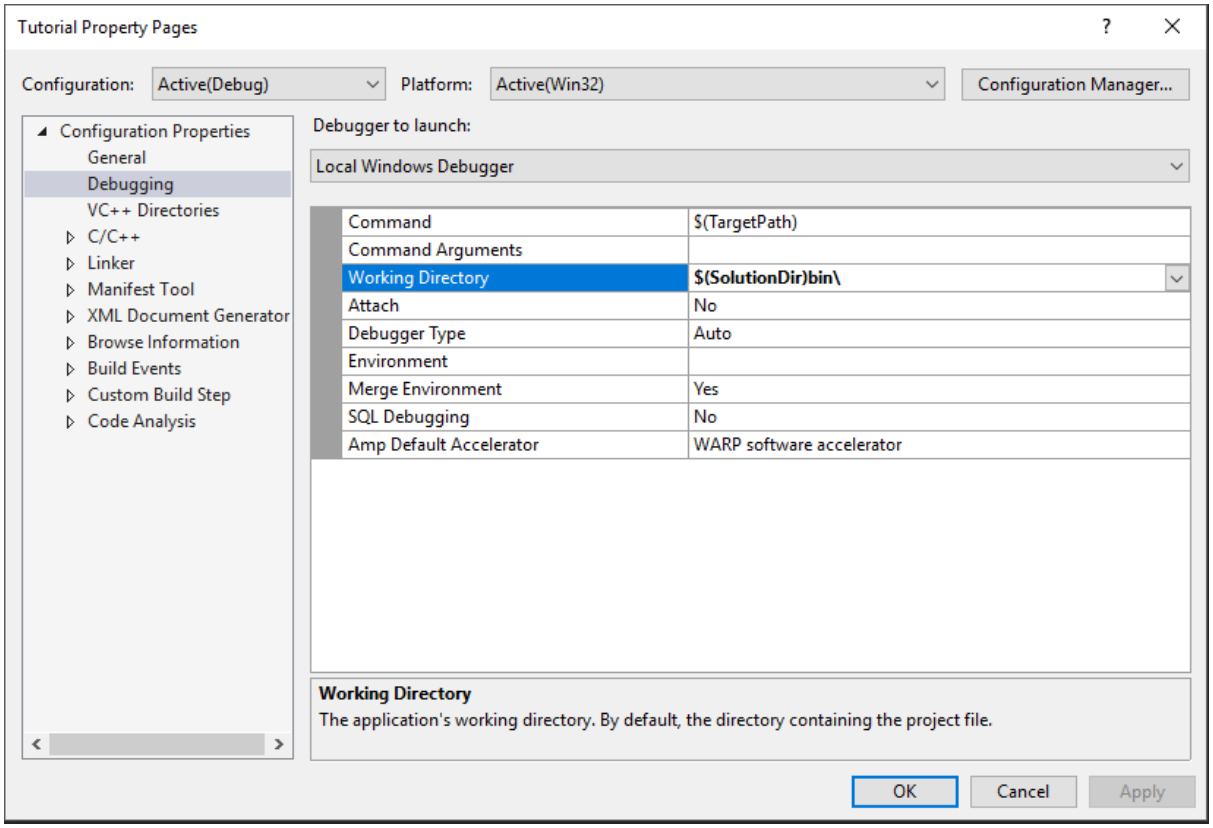
In the *Solution Explorer*, right-click on your new project and select *Set as StartUp Project*. This means that when you press the *Compile* button, this is the project that will be compiled.



Finally, we need to set the execution path of the project.

Right-click on your new project and select *Properties*.

Change the *Working Directory* under the *Debugging* settings to **$(SolutionDir)bin**



This will ensure the program can find all required libraries and resource files when it executes.

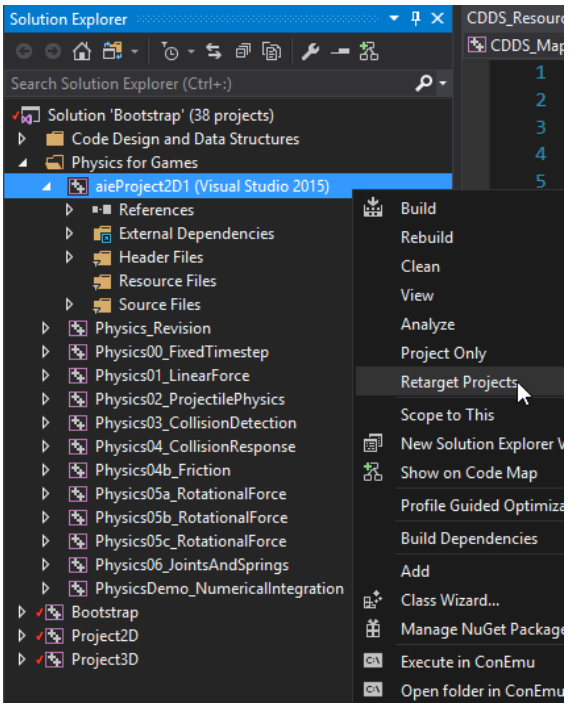
You've just created a new project that uses the AIEBootstrap library.

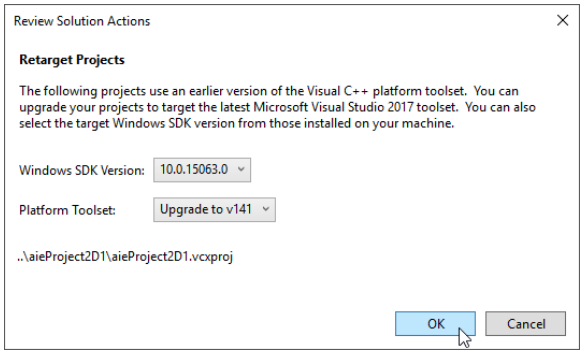
Retarget Your Project:

Before continuing with the rest of the tutorial, try to compile and run your project now.

If you get an error saying your project was made using a different version of the compiler, you'll need to retarget the project. (This is because the templates have been made with a previous version of Visual Studio).

In the *Solution Explorer*, right-click on the project name and select *Retarget Projects*. Hit *OK*.





Your project will then be set to use the latest SDK, and you can compile and execute as normal.

Working with 2D Gizmos:

In a 2D project, the AIE Bootstrap framework is set up to work with textures by default.

To draw shapes like axis-aligned bounding boxes, lines, rectangles, and circles, we need to initialize the *Gizmos*.

The *aie::Gizmos::create()* function creates an instance of the *Gizmos* singleton class, and initializes the OpenGL buffers used for rendering. Without calling this function at the start of our program, none of our 2D shapes will be rendered to the screen.

Add the following *#include* statement to the top of your application .cpp file:

```
#include "Physics_RevisionApp.h"
#include "Texture.h"
#include "Font.h"
#include "Input.h"
#include <Gizmos.h>
```

In your application *startup()* function, add the following line of code:

```
bool Physics_RevisionApp::startup() {
    // increase the 2d line count to maximize the number of objects we can draw
    aie::Gizmos::create(255U, 255U, 65535U, 65535U);

    m_2dRenderer = new aie::Renderer2D();

    // TODO: remember to change this when redistributing a build!
    // the following path would be used instead: "./font/consolas.ttf"
    m_font = new aie::Font("../bin/font/consolas.ttf", 32);

    return true;
}
```

To draw in 2D, we'll also set up an orthographic projection matrix. This will be created by passing in some arguments to the *glm::ortho()* function. The return value of this function is our required projection matrix.

In order to use *glm::ortho()* we'll need to include the GLM extension header, *glm/ext.hpp*. Add the following include definition to the top of your application .cpp file:

```
#include "Physics_RevisionApp.h"
#include "Texture.h"
#include "Font.h"
#include "Input.h"
#include <glm\ext.hpp>
#include <Gizmos.h>
```

The matrix needs to be passed to the *aie::Gizmos::draw2D()* function, which renders all the gizmos that have been added to that point using the *aie::Gismos::addX* functions.

Note that after each call to *aie::Gizmos::draw2D()* the buffers are cleared, so each time through the update loop you must re-add all gizmos via calls to *aie::Gismos::addX* to render the next frame.

In your application's *draw()* function, add the following code to create the projection matrix and pass it to the *draw2D()* function:

```
void Physics_RevisionApp::draw() {
    // wipe the screen to the background colour
    clearScreen();

    // begin drawing sprites
    m_2dRenderer->begin();

    // draw your stuff here!
    static float aspectRatio = 16 / 9.f;
    aie::Gizmos::draw2D(glm::ortho<float>(-100, 100,
                        -100 / aspectRatio, 100 / aspectRatio, -1.0f, 1.0f));

    // output some text, uses the last used colour
    m_2dRenderer->drawText(m_font, "Press ESC to quit", 0, 0);
    // done drawing sprites
    m_2dRenderer->end();
}
```

We're now ready to draw some shapes.

Our *Breakout* clone will have a 10 x 5 grid of blocks, a ball, and a paddle.

For the moment, we'll add the gizmos for these shapes in the application's *update()* function. We'll hard-code the position of each object for now, but in a real implementation we'd want to create classes representing each shape and ensure each object is updated and drawn via calls to the appropriate class function (which is actually what we'll do in future tutorials).

Add the following code to your application's *update()* function:

```
void Physics_RevisionApp::update(float deltaTime) {

    // input example
    aie::Input* input = aie::Input::getInstance();

    static const glm::vec4 colours[] = {
        glm::vec4(1,0,0,1), glm::vec4(0,1,0,1),
        glm::vec4(0,0,1,1), glm::vec4(0.8f,0,0.5f,1),
        glm::vec4(0,1,1,1)
    };

    static const int rows = 5;
    static const int columns = 10;
    static const int hSpace = 1;
    static const int vSpace = 1;

    static const glm::vec2 scrExtents(100, 50);
    static const glm::vec2 boxExtents(7, 3);
    static const glm::vec2 startPos(
      -(columns >> 1)*((boxExtents.x * 2) + vSpace) + boxExtents.x + (vSpace/2.0f),
      scrExtents.y - ((boxExtents.y * 2) + hSpace));

    // draw the grid of blocks
    glm::vec2 pos;
    for (int y = 0; y < rows; y++) {
        pos = glm::vec2(startPos.x, startPos.y - (y* ((boxExtents.y*2) + hSpace)));
        for (int x = 0; x < columns; x++) {
            aie::Gizmos::add2DAABBFilled(pos, boxExtents, colours[y]);
            pos.x += (boxExtents.x*2) + vSpace;
        }
    }

    // draw the ball
    aie::Gizmos::add2DCircle(glm::vec2(0, -35), 3, 12, glm::vec4(1, 1, 0, 1));
    // draw the player's paddle
    aie::Gizmos::add2DAABBFilled(glm::vec2(0, -40), glm::vec2(12, 2),
                                 glm::vec4(1, 0, 1, 1));

    // exit the application
    if (input->isKeyDown(aie::INPUT_KEY_ESCAPE))
        quit();
}
```

We start by declaring some static constant values that will be used to position and draw our objects on the screen. The start position of the top-left block is calculated relative to the centre of the screen (which has the coordinates 0,0).

We then use a nested loop to add the 10 x 5 grid of blocks. For this, the *add2DAABBFilled()* function is used. This will add a gizmo for a filled, 2D axis-aligned bounding box to the gizmo buffers. The *filled* variant of this function will draw a box that is filled with a solid colour. There is also an unfilled variant available using *add2DAABB()*.

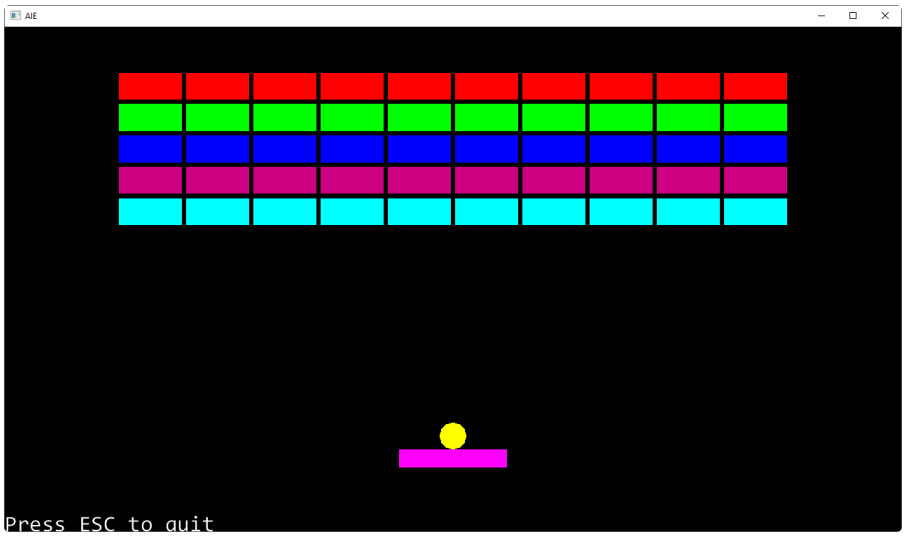
You may notice that the *add2DAABB()* function defines an argument called *extents*. The first argument sets the centre of the box, and the *extents* argument specifies the half-width and half-height (i.e., the distance along the x or y-axis from the centre of the box to it's edge). It's important

to remember that the width and height of the box will actually be twice as big as the values specified for the extents.

After the block grid is created, we add the gizmos for the ball (a 2D circle), and the player's paddle (another AABB).

All these gizmos are added to the buffers defined within the *Gizmo* class. When we call *aie::Gizmos::draw2D()*, this data is copied to the OpenGL rendering buffers and the screen is updated. So although we are adding our shapes during the *update()* function, nothing will appear until *aie::Gizmos::draw2D()* is called in the *draw()* function.

Compile and execute your program. You should see a *Breakout*-like game screen:



Experiment with the *aie::Gizmo* class by adding some more 2D shapes, and changing the colours of the shapes.

You can see all the functions available in the *aie::Gizmo* class (and thus all the shapes you can draw) by either searching for and opening *Gizmos.h* in the AIE Bootstrap project, or by right-clicking on any reference to the *aie::Gizmos* class in your code and selecting *Go To Definition*.