XNA Cheat Sheet

Tim Scarfe, 15th Nov 2010

Today you are going to be set your coursework and I wanted to illustrate some important core XNA concepts that will allow you to do some really cool stuff. Some of these concepts are essentially solutions to the challenges you have been set previously, some are new things entirely i.e. keyboard interactivity. Some of you will already know about these things; in which case don't worry. I'm not going to get into the nitty gritty -- just give you the basic ideas you need.

Record State Information inside Classes

Before you add interactivity or animation, your objects need to have state i.e. how tall, position (x,y,z), colour, rotation speed, scale size etc. Usually these will be a data type of **float** (which is a number i.e. 1.11, 3.4. 4000.55). **Vector3** which is a 3 dimensional number, or possibly a **Color**. C# has the concept of **private** and **public** members on classes, and also **getters** and **setters** but for now don't worry about that stuff. Just do this for properties on classes:

public float Size = 100;

- public means you can access it from the outside
- float is the data type (replace with **Vector3**, **int**, **bool** etc)
- Size is the name of the property
- 100 is the default value it takes, before you **override** it on the **instance** of the class from outside (remember the cubes from last week, we overrode the default size of each from the **Game** class)

Another example:
public bool IsDrawn = false;

Animation

Think of animation as being interpolation against time. What you are doing is taking any property (as described above) and changing its value over time (as a function of time). Animation implies we are changing the position of an object but actually you can animate its colour, size, anything!
If there is a linear dependency between time elapsed and the property you are changing, it's a linear animation (looks constant). If it's a non-linear animation i.e. with some acceleration it will look like it's "zipping". You could easily do this by using your Bezier functions you learned about to set the dependency between your property and time, because Beziers are

third degree functions there will naturally be some “zippiness” to the resulting animation! Third degree means when you plot them they have 2 kinks in the curve -- but you should already know that! 4th degree means 3 kinks etc. Most non-linear animations just have one kink i.e. a parabola (second degree function y=x^2).
Animations are always done in the **Update** method. All you have to do is change a class property as a function of time.

Examples:

```
/// this updates the X component of the _currentPosition property over time
public void Update(GameTime gameTime)
{
   _currentPosition = new Vector3(
      _currentPosition.X + (float) gameTime.ElapsedGameTime.TotalSeconds,
      _currentPosition.Y,
      _currentPosition.Z
      );
}
///  change the colour of an object in a sinusoidal way over time
/// note Color.Lerp interpolates between two colours, 3rd argument is
/// on the interval [0,1]
public void Update(GameTime gameTime)
{
   Colour = Color.Lerp(
            Color.Black,
            Color.Green,
            (float)Math.Sin((gameTime.TotalGameTime.TotalSeconds/2)+0.5)
         );
}
```

Note that sometimes C# functions need a float, and sometimes we have a double, so we can cast it into a float by putting in a **(float)** before. Also be careful of integer division. 3/2 will equal 1 (no remainder) in C# because the result is an Integer. 3d/2 or (double) 3/2 will be what you expect i.e. 1.5d.

Homogeneous Matrix Transformations

These happen in the Draw method on all objects.

- They are executed in back-to-front order
- Matrix.Translate() will set the origin somewhere else i.e. for rotating an object around a point, or finally to set the position of an object
- Matrix.Scale is useful for scaling objects. Remember that when you set a scale of say 2, the coordinate system is **doubled** for that object so a subsequent translation will move the object twice as far. Everything is arbitrary.
- Matrix.CreateRotation[X,Y,Z] is for rotating objects

- Homo matrixes are great because you are always multiplying, never adding. To give you an illustration of why this is good -- say you were working on a plane in 3d space and you wanted to add some propellers. You already have a matrix that represents the position, scale, orientation of the plane etc -- called mat_plane. All you need top do is sort the relative position etc of the props. So you multiply them out prop1 = mat_plane * Matrix.Translation(50, 0,0) i.e. prop1 is where the plane is, but shifted along by 50 on X.
- So how do we solve the cube around the flagpole thing from last week?

Cube around the flagpole

1. Translate to the flagpole (set the origin to what we want to rotate around)
2. Perform rotation
3. Translate away from the flagpole to the expected final place
4. Scale the object to the correct size

Remember you have to do these things in back to front order -- these are matrix multiplications!

Keyboard Interactivity

In the Update method, we can add interactivity from the mouse and Keyboard. This is critical for any kind of interactivity.

```
public void Update(GameTime gameTime)
{
   KeyboardState ks = Keyboard.GetState();
   if( ks.IsKeyDown(Keys.Left) ) {

      _position += (float) gameTime.ElapsedGameTime.TotalSeconds;
   }

   MouseState ms = Mouse.GetState();

   if (ms.LeftButton == ButtonState.Pressed)
   {
      // do something
   }
}
```

Use Visual Studio intellisense to inspect the properties and methods on MouseState and KeyboardState. It’s really simple and will make your game 100 times more interesting.

Change the Camera Position

In the templates I have given you the camera position is fixed. You can change it by overwriting the **_view** matrix in the Game class. All you need to do is use **Matrix.CreateLookAt()** to make a new Matrix.The arguments for CreateLookAt will let you set positions for the camera, where it's looking at, and it's orientation (typically **Vector3.Up**). Imagine the cool possibilities, you could have a car running around a race track and the camera chasing it from distance with some cool fudging back and forth like in racing games. You can also play with the _projection matrix, this sets the field of view and near/far clipping planes. Think of the projection matrix as being a pyramid with the top chopped off, this is your 3d viewing space which is flattened to your 2d monitor.

Draw 2D Text or Images on the Screen

Text - http://msdn.microsoft.com/en-us/library/bb447673.aspx
Images (using SpriteBatch) - http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series2D/Drawing_fullscreen_images.php

Load in an external model and draw it

You can see examples of this in my Cube class from last week -- you need to load the image into a Model object in the LoadContent method, then draw it in the Draw method.