Please read all of this before coding.

About the Software

This "game of games" is designed to package several **single-player, arcade-style, score-based** games into one so that it is compatible with the Xbox 360 and Xbox Live. It is set up so that you will not need to deal with screens, low-level controller inputs, Xbox 360 Gamer Profiles, or storage. You can concentrate all of your efforts completely on making the game, so that the gameplay and game quality are as good as they possibly can be.

System Requirements

- Windows
- Visual Studio 2010 and XNA 4.0
- **NEW**: Games for Windows LIVE (you must sign into the game with a "Guest" account, which you can create from the Games for Windows LIVE menus by pressing Home after starting the game.)

What NOT to Change

Any file that says "You should not change this!" in fancy letters at the top does not need changed. Many will be changed later to make the software prettier and more user-friendly.

Please do not add any files (or change any files) to any folder other than the YourGameNameHere folder in the code project and the YOURGAMEHERE folder in the content project.

What to Change

- You will need to create a class which extends **GameScreen**, which will be the entry point for your game code. The constructor of this will be called when the player selects your game from the main menu. This will have an Update, Draw, and Reset method as well. You must call the Reset method at the end of your constructor code, and when the player chooses to replay it, Reset will be called and not the constructor. So put all variables and objects that need to be initialized (Lists, etc) in your constructor, and set all values that do not change in your constructor, but set all values that do change in the Reset method. You can copy StupidGameScreen.cs to the YourGameNameHere folder (after renaming the folder) and work from the code in that file (rename that file to "<YourGameName>Screen" as well). This should provide a good example of the methods used. There is a "GameOver" method, which you should call once when the game is over (the player dies, timer runs out, etc.).
- In the **Loader** class, rename the **LoadYOURGAMENAMEHERE Content** method appropriately and use this to load the content for your game. Additionally, in the Loader class, add your game info (just like the other sample game infos) to the **LoadGameInfo** method. In this, you will need a short (250 characters or less) description of the game along with the Title and Authors. You will also need a box art, which is a 584 by 700 JPEG image. Add this box art to Content/Shared/BoxArt and then load the Texture from the **LoadShared** method in Loader, and pass whatever you named the Texture to the game info for your game. This is done already with the other sample game infos, so you can look at that code.
- In the Content project, don't add anything other than your box art to the "Shared" folder. Add everything to the YOURGAMEHERE folder (after renaming it to your game name).

Using the Controller Class

Games for Xbox Live are required to work with any controller configuration, so for example, you cannot create a game that works only with the "player index one" controller, as someone might want to play with the second controller that is plugged in. But you don't need to worry about all that, because the **Controller** class, along with the "Press Start" screen, handles it for you. Just use **GameWorld.controller** and you're all set. This has the following methods that will be useful:

- bool GameWorld.controller.ContainsBool(ActionType action)
    - Use this to check if a "button" is being pressed or Boolean action is being performed from the controller. If a non-Boolean ActionType is passed, then this returns true if it is not 0 and false otherwise. The following are ActionTypes should be passed to this method.
    - ActionType.AButton
    - ActionType.BButton
    - ActionType.XButton
    - ActionType.YButton
    - ActionType.AbuttonFirst (Only returns true when unpressed on last update.)
    - ActionType.BbuttonFirst (Only returns true when unpressed on last update.)
    - ActionType.XbuttonFirst (Only returns true when unpressed on last update.)
    - ActionType.YbuttonFirst (Only returns true when unpressed on last update.)
    - ActionType.DPadDown
    - ActionType.DPadUp
    - ActionType.DPadLeft
    - ActionType.DPadRight
    - ActionType.LeftBumper
    - ActionType.RightBumper
    - ActionType.LeftBumperFirst (Only returns true when unpressed on last update.)
    - ActionType.RightBumperFirst (Only returns true when unpressed on last update.)
    - ActionType.Pause
    - ActionType.Select
    - ActionType.SelectionDown
    - ActionType.SelectionLeft
    - ActionType.SelectionRight
    - ActionType.SelectionUp
- float GameWorld.controller.ContainsFloat(ActionType action)
    - Use this to check if a floating-point value based action from the controller is performed. If a Boolean-based ActionType is passed, this will return 1, and otherwise 0. The following ActionTypes should be passed to this method.
    - ActionType.LeftTrigger (0 when released, 1 when fully pressed, and anywhere in-between.)
    - ActionType.RightTrigger (0 when released, 1 when fully pressed, and anywhere in-between.)
    - ActionType.LookHorizontal (The x-value of the right trigger, in the range [-1, 1].)
    - ActionType.LookVertical (The y-value of the right trigger, in the range [-1, 1].)
    - ActionType.MoveHorizontal (The x-value of the left trigger, in the range [-1, 1].)
    - ActionType.MoveVertical (The y-value of the left trigger, in the range [-1, 1].)
- void GameWorld.controller.Rumble(float leftMotorAmount, float rightMotorAmount, long durationInTicks)
    - Sets the rumble of the controller. I recommend using this for effect, but not overusing it. The left and right motor values must be in the range [0, 1], and the duration is in ticks (10,000,000 ticks per second).
- You don't need to use any methods other than ContainsBool, ContainsFloat, or Rumble.

Testing Your Game

It is extremely important to test your game, because it cannot have any bugs. Playtesting is always good, but you should go beyond that. I suggest that you get an XNA student account from MSDNAA and Dreamspark so that you can play the game on the Xbox, to make sure it doesn't lag or have any other issue that you might not have been able to reproduce on the PC.

In addition, be sure to observe the "tile safe area" of the screen. Older TV's (and some newer) have overscan issues which cut off the edges of the screen, so if you have any important stuff there (such as the player's character, HUD, etc.), some players may not be able to see it, and that is bad. When you run the program in "Debug" you will see a semi-transparent border in which you should not draw important stuff. You should still draw in the entire screen rectangle

(which is always 1920 by 1080), but do not draw anything that is crucial to gameplay, such as the player, HUD, score display, or anything else which you think might be crucial. Just use your best judgment as to what is crucial.

Particles

As you know, particles are awesome, and you should highly consider using them in your game. They give a game that extra visual spunk that will make you look like a game dev pro instead of a game dev noob. For your convenience, your wonderful and talented TA has written and attached a particle engine to the solution. Here is how to use it.

- All classes related to particles are in the Particles folder.
- The ParticleContainer class is used to hold all the particles. The particles that it holds need not be the same type of particle (a particle type is specified by a ParticleAction, see below). All screens have a particle container, so your game will have one automatically, but it will need to be constructed in your game's constructor. The construction takes one argument—the maximum number of particles allowed. I recommend 1,000 so that you can be sure it doesn't lag. You might consider having a single particle image consist of several smaller "particles". This is updated and drawn for you, so don't call these methods.
    - You can use the Add method to add a new particle. When you do, pass the particle's initial position (x, y), velocity (x, y), Color object (it is recommended that you do not create a "new" Color each time, but rather store the color you want to use or just use a built-in color such as Color.White), the scale, the rotation, the action unique identifier and the action. These last 2 will be described later.
    - The Reset method is used to clear all particles without doing anything to memory. You should use this when you Reset your game.
    - The ApplyAcceleration method can be used to apply an acceleration to all of the particles at once. Note that a particle does not store an acceleration vector, so it must be stored separately if a particle is going to have a 2^{nd}-positional derivative. In the background screen example (included in the solution), a "wind" is applied to show how this method can be used.
- You should not actually create a "new" Particle yourself, as this is what the ParticleContainer does. It handles memory very efficiently, so use the ParticleContainer.Add. A particle has all of the attributes described in the "Add" method of ParticleContainer above. When you want to get rid of a particle, call its "Dispose()" method. The particle container will take it from there.
- A ParticleAction is what is used to perform an operation on a specific type of particle. Because the ParticleContainer can hold several different types of particles, a ParticleAction class is used to distinguish between the types. A ParticleAction object is passed to ParticleContainer when creating a particle, but this object that is passed should be the same, shared object that is passed when creating all of the particles of that type. In other words, DO NOT create a new ParticleAction object every time you call the Add method of ParticleContainer! The TechnicolorAction is a good example of a ParticleAction. This holds a single, static instance which is created once and passed to the Add method. Note that because all particles share the same object, there are no "local" variables belonging to the ParticleAction. This is why the Update and Draw method for the ParticleAction take a Particle as a parameter. You can get and set all of the public variables of a particle in the Update method, and use these to draw the particle in the Draw method. You will probably want to copy and work off of the TechnicolorAction class when creating a new type of particle.
    - Note that when a particle is created, it sets a "CreationTime", which is the GameClock.Now value when it was created. This is very useful in determining what attribute it should have from the Update method of the ParticleAction it is using. For example, if you know the particle exists for a maximum 1 second, you can linearly interpolate based on the difference between the current time and creation time, divided by 1 second, to get a value between 0 and 1, and use this to set things like scale, color, etc. The TechnicolorAction.Update method does this, so see that example.
    - Each particle has an "action unique identifier." This is not used in the TechnicolorAction example, but if you want to have a ParticleAction that, say, draws different textures, you may want to use this value to identify the texture to draw. This is merely for convenience. You do not have to use this because you can always create a separate ParticleAction.
- Although you can call the Add method of ParticleContainer directly, a ParticleEmitter interface is included so that you can make your own particle emitter to emit particles however you like. A CircularParticleEmitter is

included as an example. The Update method of this (and all emitters) takes a Vector2 as the position in which to launch the next particle. When the CircularParticleEmitter is constructed, it takes as a parameter the delay (in ticks) between particles launched, the maximum velocity, the action corresponding to a particle that will be launched, and the particle container. When it launches a particle, it will pick a random angle and a random velocity less than the maximum specified, thus emitting a circle of particles. Note that more than one particle can be emitter per frame if the delay is less than the frame refresh rate, and if so, the circular particle emitter will launch multiple particles, linearly interpolated between the current position and the previous. If you create your own type of particle emitter, keep this linear interpolation in mind, as it is very important to create a smooth looking particle stream.

- A ParticleBouncer object is included as well. This extends CircularParticleEmitter, and it basically just bounces around the screen like the Pong ball. The BackgroundScreen class is what holds a ParticleBouncer, which results in what you see when you start the game. Note that everything needs updated, but only particles need drawn. Don't draw or update the particles or the particle container yourself, because the Screen will do that automatically.

Additional Notes

- For all of the classes that you add, make sure that the namespace begins with Project290.YOURGAMENAME so that when all the files are merged, there will not be conflicts.
- The resolution will always be 1920*1080. This will be the case regardless of the actual monitor's or TV's resolution, so you can assume that the width and height will always be 1920 and 1080, respectively. So for example, if you want to draw something to the center of the screen, drawing to (1920, 1080) / 2f will be the center. Be sure to keep in mind the "tile safe area" (see "Testing Your Game").
- There is no need for you to use the storage interface at all. The software is set up so that the GameScreen class has an int, "Score", that can be set. You should use this to set the player's score during game play. For example, when the player kills a space zombie emu penguin blob, add x to Score. That's actually all you have to do. Once the game ends, the player will be shown the score, and the score will automatically be added to the high scores table (if higher than the previous).
- You do not need to make any screen other than your game screen—so no title, pause, or game over screen. All games will share the same screen format for the title, pause, and game over screen, which looks bad now, but will be awesomized later.
- Layer depths now work independently between screens (as opposed to the Pong code). You can use layer depths in [0, 1] in your game screen without fear that it will affect the pause or game-over screens.
- The Farseer Physics is included in the solution. I recommend that you try using it if you're doing anything more advanced in terms of physics. **Note that the "World" object that holds all of the physics is included in the Screen object, so your Game Screen will automatically have this, but it will still need constructed in your Game Constructor.** The world object is updated for you in the Screen class. It's very easy to use…check out the Farseer simple samples uploaded to the Blackboard site.
- A "sprite sheet" of the Xbox 360 controller buttons is included in Content/Shared/Fonts. You should use one of these buttons for your "how to play" image (a 1450*800 PNG), and for any in-game display of the button. To display a button, call Drawer.DrawControllerSymbol(…) and pass the button type (with the Buttons enum) and all other drawing info.
- Please try to keep the size of your game less than 15 Megabytes, if possible. To find out the size of your game, right-click on your project and select "Package as XNA Creators Club Game", and then look in the bin folder to find a .ccgame file—the size of this is the size of your game. If you absolutely must go over 15 MB, that is okay, but please try to keep the game as small as possible, as people who own Xbox 360s with limited hard drive space will not be able to download the package if it is too large overall.

As always, ask questions if you are unsure about anything!