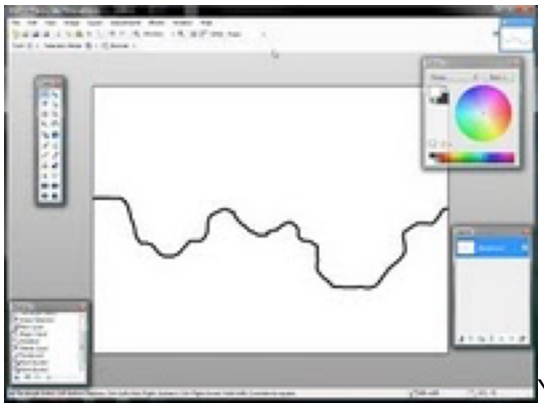
Destructible Terrain

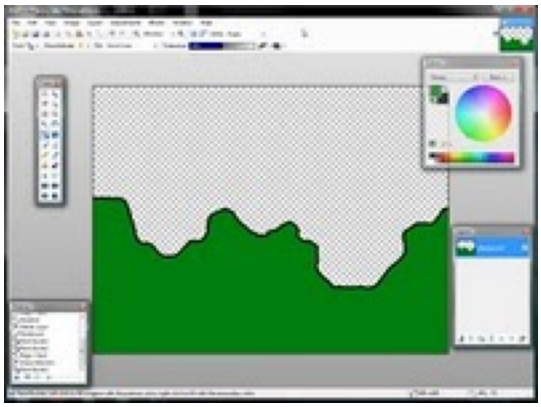
Today I'm going to give you a quick overview on how to create deformable 2D level (similar to what is seen in games like Worms and Lemmings)

First I will create a new XNA 2.0 project, next I'm going to create a level using Paint.Net .

I'll create the new image with a size 800 (w) x 600 (h) as this is the default size a XNA 2.0 project uses. Now we are ready to start drawing the level, select the Paintbrush tool and change the Antialiased button to Antialiasing Disabled, then draw your outline of your level (preferably in black).

 You should now have something similar to this, from here select the top section (above the black line) with the Magic Wand tool, and press the delete key. The top section should now change to grey and white checkered boxes, this represents the alpha layer.
Choose your desired level colour and then select the Paint Bucket tool and click on the bottom sectin of your level.

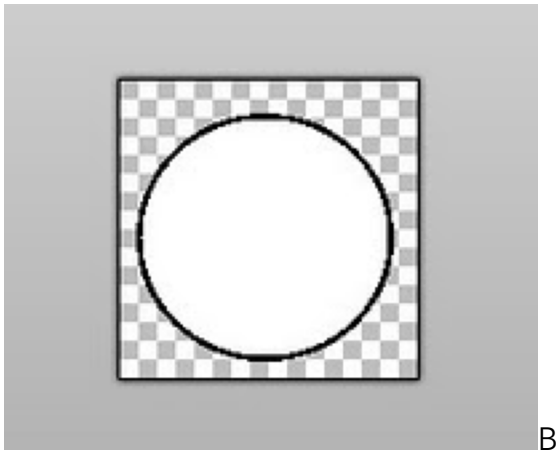
You should now have something along the lines of the following:

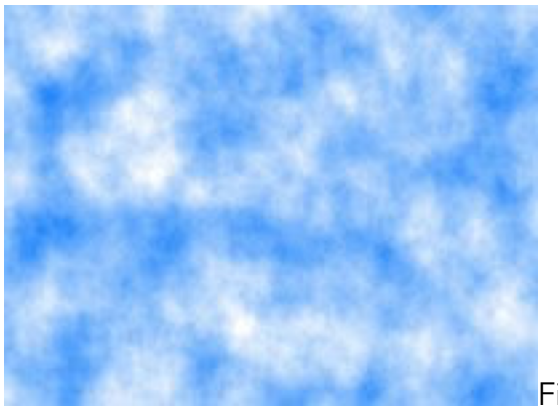


From here you are ready to save, click Save As and select the "*Save as type:*" to "**PNG (**.png)*" and call it "level"
(click here to download a copy of my level)

So thats our level done, now lets create our deform image, we do this in a very similar way as the level so I wont go threw it step by step.

Im going to make the deform image 128(w) x 128(h), basically its going to be an circle (so i'll draw it using the Ellipse tool using the same Brush width and Anti-Aliasing options as I used for the leve.

 Because we are going to be using this image as the image that will deform the level we need to specify that the iniside of the level is white (we will replace this in code with alpha pixels) and the outside of te ellipse will be transparent / alpha like this. Then save this as a PNG file as well and call it "deform". Or you can download my copy here .

 Finally lets create a nice sky background, create a new image 800x600, then select a nice blue colour for your Primary color and make leave your secondary colour white. Then simply select from the Effects menu "*Effects* ➜ *Render* ➜ *Clouds*"and clickthe *OK*button. Save this image as sky.jpg (Note! We don't need to save this as a PNG file as the sky image contains no alpha layer, so by using a JPG format we can save a little on disk space)

Right, now we are ready to begin coding.

Drag & Drop the above two png files and the sky.jpg you created into you content folder in your project.

Now open the Game1.cs file and lets add the following below the following line:
SpriteBatch spriteBatch;
We need to declare the sky, level, and deform sprites:

```
private Texture2D textureSky;
private Texture2D textureLevel;
private Texture2D textureDeform;
```

Then in the LoadContent class lets replace the TODO comment by loading our content:

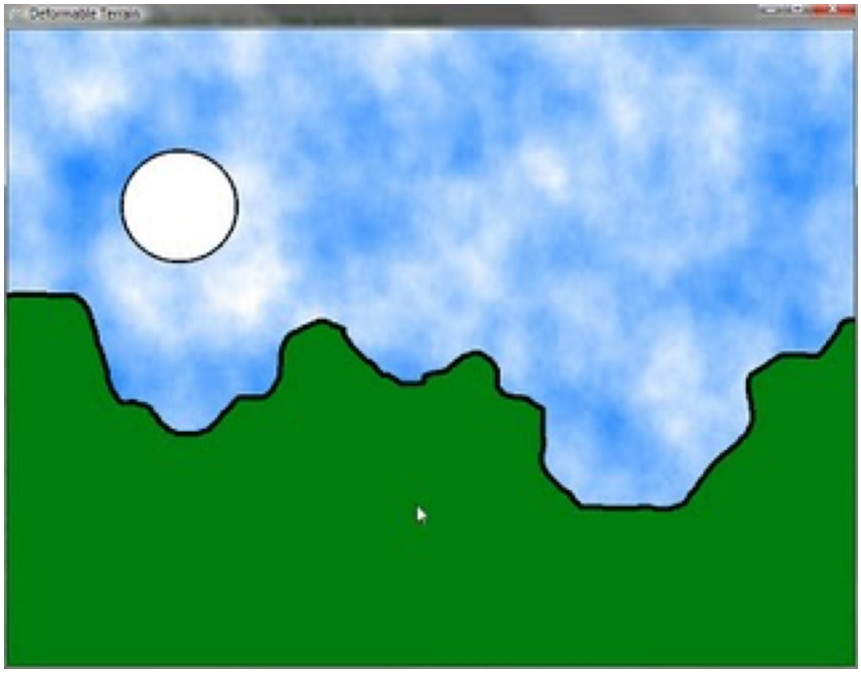
```
textureSky = Content.Load<Texture2D>("sky");
textureLevel = Content.Load<Texture2D>("level");
textureDeform = Content.Load<Texture2D>("deform");
```

We are already ready to start on the draw class, so lets replace the TODO comment in the Draw function with the following:

```
spriteBatch.Begin();

spriteBatch.Draw(textureSky, new Vector2(0, 0), Color.White);
spriteBatch.Draw(textureLevel, new Vector2(0, 0), Color.White);
```

```
spriteBatch.Draw(textureDeform, new Vector2(100, 100), Color.White);

spriteBatch.End();
```

Right, already we are ready to build the game and see our level, it should look something like this:



Not bad for just 11 lines of new code!

Moving the deform sprite around

Too move the deform sprite we are going to use the mouse. So if you run the game now you will notice that the mouse cursor is hidden over the game window. Lets change that to make is visible.

In the Initialize function replace the TODO comment with the following line of code:

```
this.IsMouseVisible = true;
```

Next we declare the Vector2 variable to store the mouse position (add this below the where we added the Texture2D declarations) and also declare the current mouse state:

```
private Vector2 mousePosition;
private MouseState currentMouseState;
```

Now we need to update the mousePosition, so lets create a new function do this for us:

```
protected void UpdateMouse()
{
```

```
currentMouseState = Mouse.GetState();

 This gets the mouse co-ordinates
 relative to the upper left of the game window
mousePosition = new Vector2(currentMouseState.X, currentMouseState.Y);
}
```

We then replace the TODO: comment line in the Update function with a call to the UpdateMouse function we just created.

```
UpdateMouse();
```

And finally modify the draw call for the textureDeform sprite to use the mouse position:

```
spriteBatch.Draw(textureDeform, mousePosition,Color.White);
```

Hit F5 and try it out.

Deforming the Level

In order to visually deform the level we need to grab the texture data of the level and put it into an array, this will then hold the uint value for each co-ordinate of the texture, we then modify the array values of the section of the level where the user "pastes" the deform image down, by setting the level array to equal the deform sprites texture array. And finally we update the level texture with the new array of values.

So lets first load the deform pixel array, so below where you declared the currentMouseState add the following line:

```
private uint[] pixelDeformData;
```

Then in the LoadContent function add the following after you load the textureDeform:

```
 Declare an array to hold the pixel data
pixelDeformData = new uint[textureDeform.Width * textureDeform.Height];
 Populate the array
textureDeform.GetData(pixelDeformData, 0, textureDeform.Width *
textureDeform.Height);
```

Remember because we can deform the level an endless amount of times we need to load the pixel array each time you try deform the level, so lets do that in a separate function which we can call when the mouse is clicked.

```
protected void DeformLevel()
{
 Declare an array to hold the pixel data
```

```
uint[] pixelLevelData = new uint[textureLevel.Width * textureLevel.Height];
 Populate the array
textureLevel.GetData(pixelLevelData, 0, textureLevel.Width * textureLevel.Height);

for (int x = 0; x &lt; textureDeform.Width; x++)
{
for (int y = 0; y &lt; textureDeform.Height; y++)
{
pixelLevelData[((int)mousePosition.X + x) + ((int)mousePosition.Y + y)
textureLevel.Width] = pixelDeformData[x + y * textureDeform.Width];
}
}

 Update the texture with the changes made above
textureLevel.SetData(pixelLevelData);
}
```

Before we can test it we need to actually call the function by left clicking with the mouse.

So to add mouse click support we will need to call *currentMouseState.LeftButton == ButtonState.Pressed*, the problem with this is that it would then call the UpdateMouse function every frame whilst the mouse button is pressed down. We only want to call it just once per click.

To do this we need to update the "UpdateMouse" function to keep a history of the previous mouse state, and then check that the previous mouse state is set pressed and the current mouse state is set to released (which would represent a click).

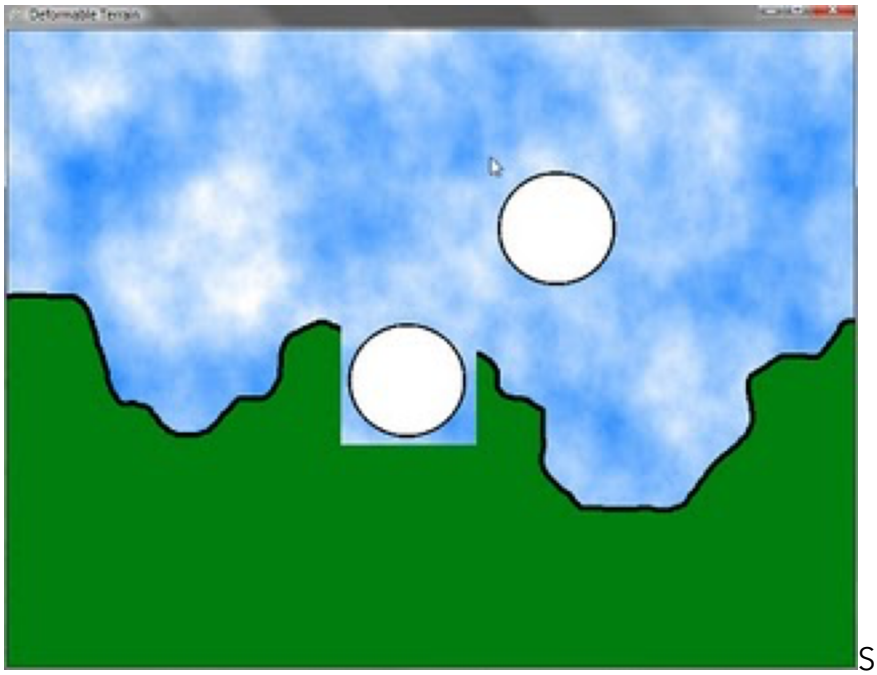
Here is the updated function with the call to DeformLevel():

```
protected void UpdateMouse()
{
MouseState previousMouseState = currentMouseState;

currentMouseState = Mouse.GetState();

 This gets the mouse co-ordinates
 relative to the upper left of the game window
mousePosition = new Vector2(currentMouseState.X, currentMouseState.Y);
```

```
 Here we make sure that we only call the deform level function
 when the left mouse button is released
if (previousMouseState.LeftButton == ButtonState.Pressed &&
currentMouseState.LeftButton == ButtonState.Released)
{
DeformLevel();
}
}
```

Now you can fire up the game and give it a try... (**Note!***we haven't done any error checking when deforming the terrain, so make sure you only click in valid areas on the level and not too close to any of the edges, else you may go outside the bounds of the array*)

You will probably end up with something like this when you click near the level:

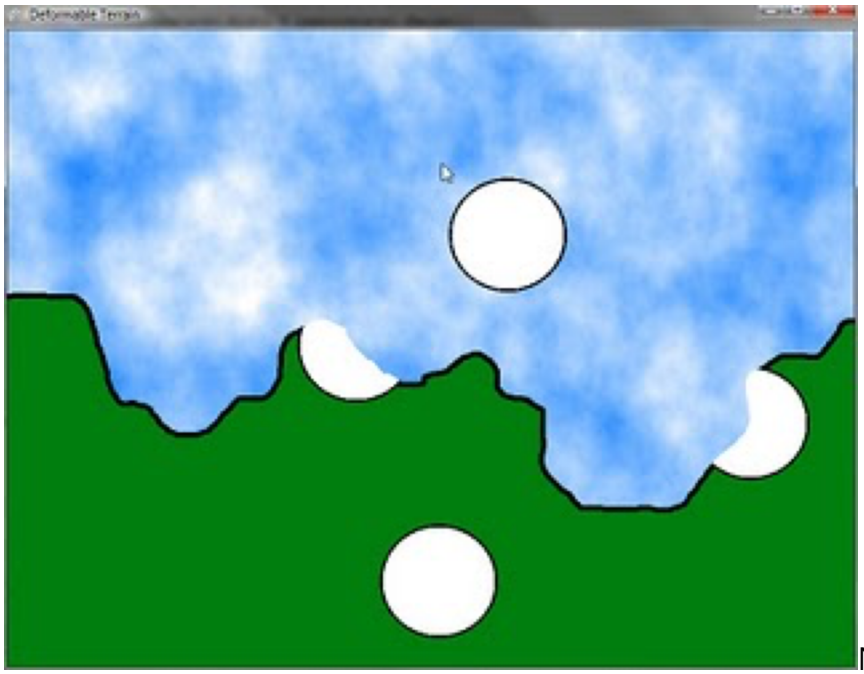


So all we have done now is updated the texture of the level with a copy of the deform level, now lets test the colour of the pixel in the deform array to see if its an alpha pixel, if it is then we don't need to update the level pixel colour, that will prevent the square alpha layer shown above around the deform sprite. We can do the same pixel test on the level array as well, this will then prevent the deform sprite from being drawn where ever the cloud is visible.

put the pixelLevelData within and if statement like so:

Here we check that the current co-ordinate of the deform texture is not an alpha value
And that the current level texture co-ordinate is not an alpha value

```
if (pixelDeformData[x + y * textureDeform.Width] != 16777215
&& pixelLevelData[((int)mousePosition.X + x) +
((int)mousePosition.Y + y) * textureLevel.Width] != 16777215)
{

pixelLevelData[((int)mousePosition.X + x) + ((int)mousePosition.Y + y)
textureLevel.Width] = pixelDeformData[x + y * textureDeform.Width];
}
```



Now if you run the game it will look something like this when you click around, which is much closer to the desired effect:

The last step is to make any pixel that is white in the deform array to alpha. And to make it a more robust we need to do some error checking to avoid any issues when deforming the level too close to the borders.

Here is the final function:

```
/ &lt;summary>
/ 16777215 = Alpha
/ 4294967295 = White
/ &lt;/summary>
protected void DeformLevel()
{
 Declare an array to hold the pixel data
uint[] pixelLevelData = new uint[textureLevel.Width * textureLevel.Height];
 Populate the array
textureLevel.GetData(pixelLevelData, 0, textureLevel.Width * textureLevel.Height);

for (int x = 0; x &lt; textureDeform.Width; x++)
{
for (int y = 0; y &lt; textureDeform.Height; y++)
{
 Do some error checking so we dont draw out of bounds of the array etc..
if (((mousePosition.X + x) &lt; (textureLevel.Width)) &&
((mousePosition.Y + y) &lt; (textureLevel.Height)))
{
if ((mousePosition.X + x) >= 0 && (mousePosition.Y + y) >= 0)
{
 Here we check that the current co-ordinate of the deform texture is not an alpha value
 And that the current level texture co-ordinate is not an alpha value
if (pixelDeformData[x + y * textureDeform.Width] != 16777215
&& pixelLevelData[((int)mousePosition.X + x) +
((int)mousePosition.Y + y) * textureLevel.Width] != 16777215)
{
 We then check to see if the deform texture's current pixel is white (4294967295)
if (pixelDeformData[x + y * textureDeform.Width] == 4294967295)
{
 It's white so we replace it with an Alpha pixel
pixelLevelData[((int)mousePosition.X + x) + ((int)mousePosition.Y + y)
textureLevel.Width] = 16777215;
}
else
{
 Its not white so just set the level texture pixel to the deform texture pixel
```

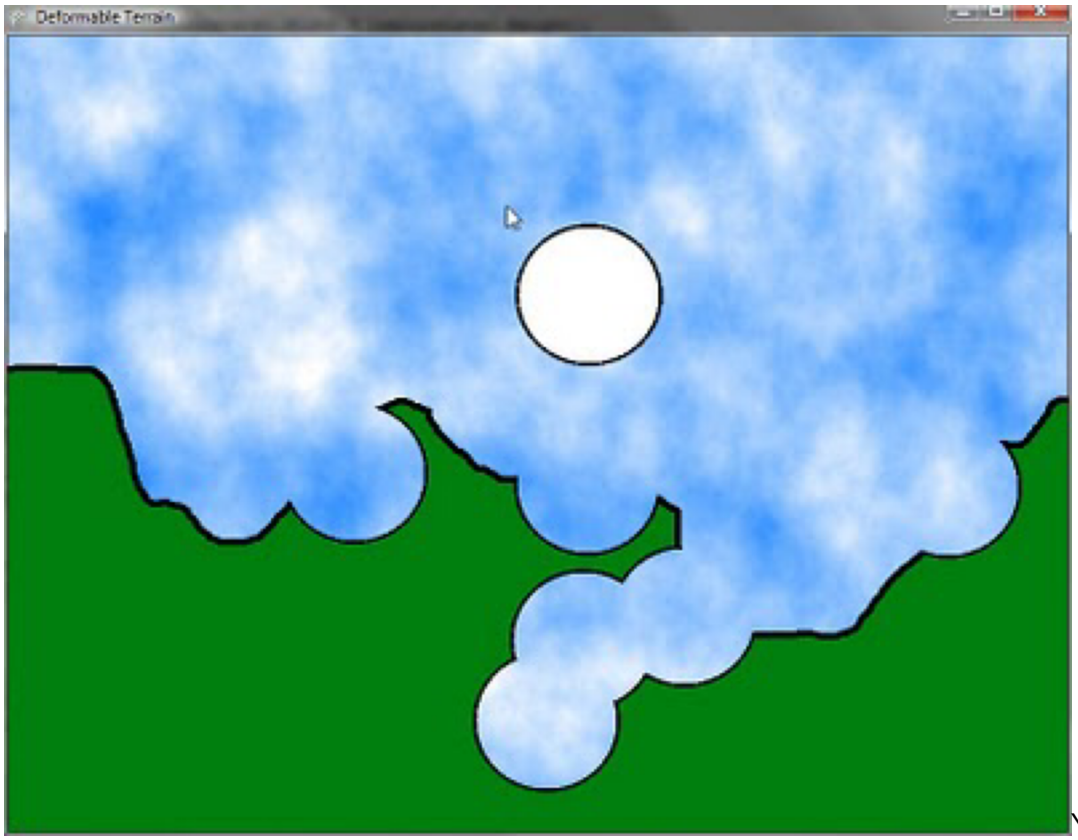
pixelLevelData[((int)mousePosition.X + x) + ((int)mousePosition.Y + y)

- textureLevel.Width] = pixelDeformData[x + y * textureDeform.Width];

}
}
}
}
}
}

Update the texture with the changes made above
textureLevel.SetData(pixelLevelData);
}

Now it should look like this (note how where the deforms occur there is still a nice black outline):



You could apply the same principles to add to the level as well, for example you could paste the dead body of a player to the terrain or a tombstone where the player died.

Here is the project source to download .

For pixel perfect collision detection you have two options, you could get the current colour value of the pixel each frame from the texture and check if it's an alpha or not (I think this would be pretty slow), or preferably you could create a bool collision array at the start of the level that sets all alpha pixels to false and level pixels to true. Then just update the collision array in the deform function where you set the level pixel to alpha.