Compositing (2D lighting pass)

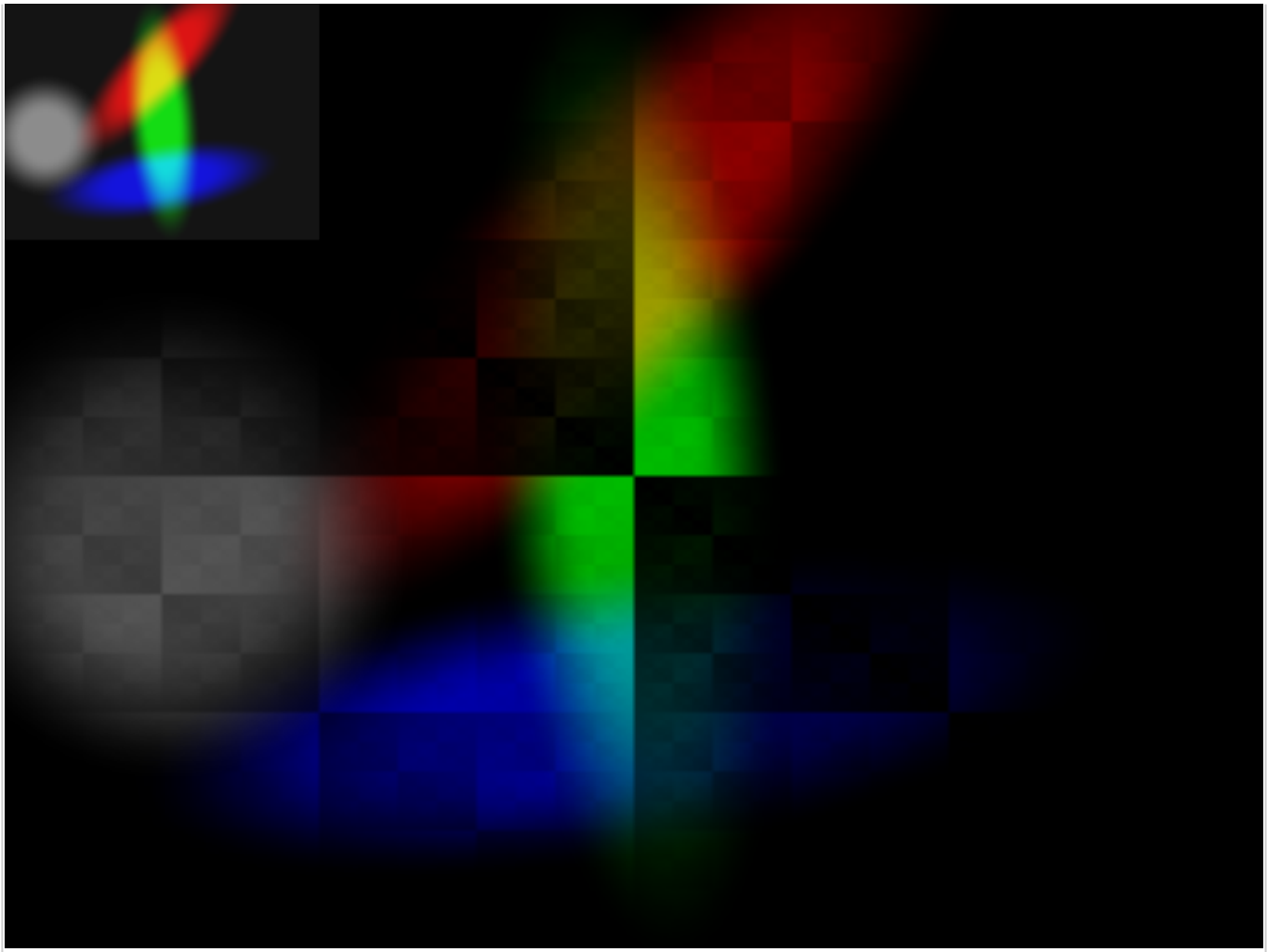
This tutorial is now outdated: there's a fully data-driven approach that can be seen in the original source material for this article.

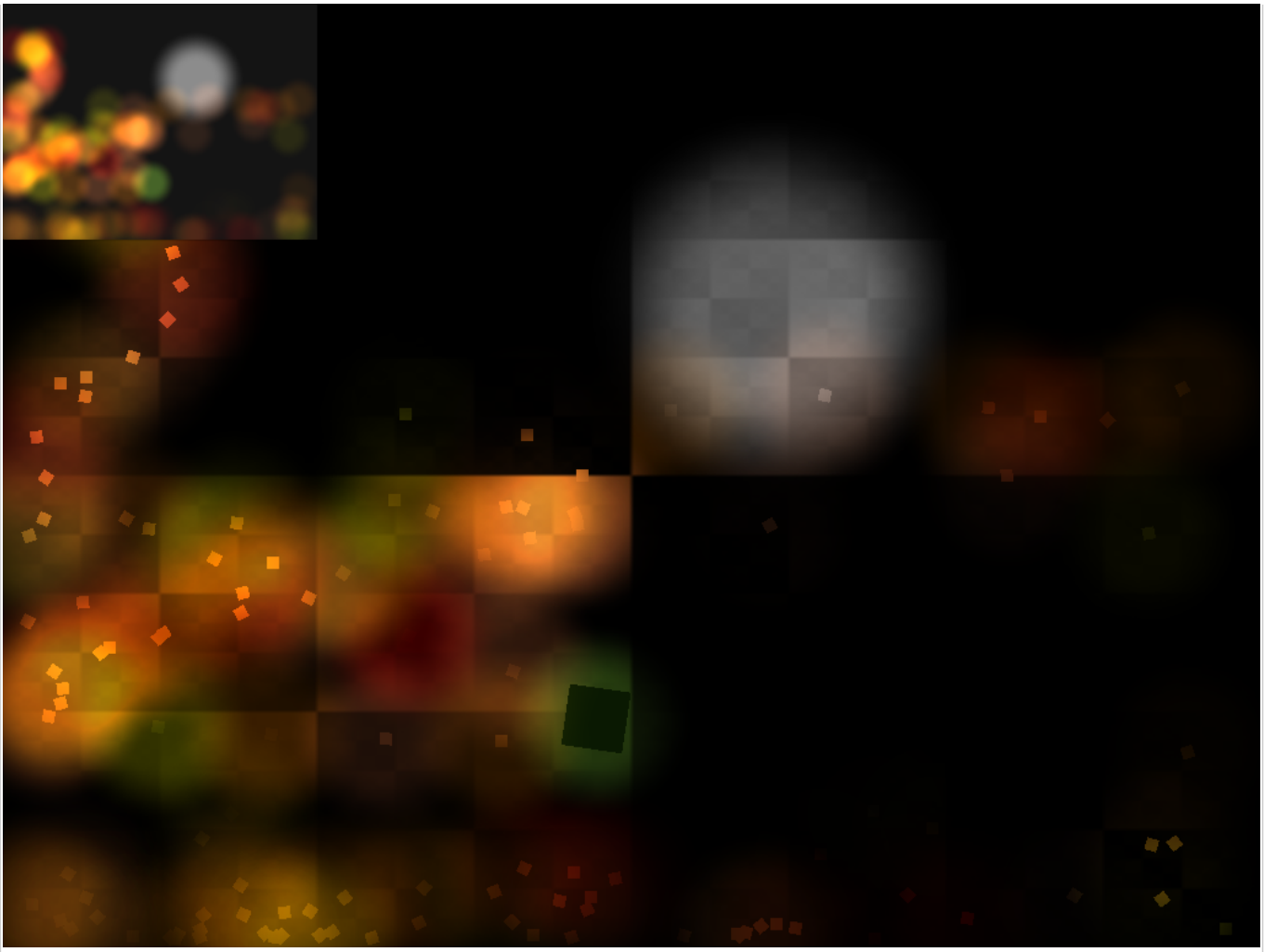
Summary

In this tutorial we'll see how easy compositing is done in orx and that it requires almost no code at all. In this simple example, we'll do some basic 2D lighting (no shadows).

Basically, we're going to first render all the lights to an offscreen texture using an additive blend mode.
We'll then use this lightmap texture in the full scene using a multiplying blend mode to simulate light/darkness.

All the files (including config, source, project and binaries) can be found on the Github public Git repository.





Code

Let's now have a look at the source code: src/compositing.cpp.

Here's the `InitTextures()` function.
That's where we're going to create a texture for our lightmap rendering but also a background and a spotlight texture.
We could have used external bitmaps for those last two but it's a good way of showing how textures can be procedurally generated.

First, we create a 256×256 texture and fill it with a XOR pattern.

```
void InitTextures()
{
  orxU32      u32X, u32Y, u32Size;
  orxFLOAT    fScreenWidth, fScreenHeight, fSpotRadius;
  orxTEXTURE *pstTexture;
  orxBITMAP  *pstBitmap;
  orxU8      *au8Data;

  // Gets screen size
  orxDisplay_GetScreenSize(&fScreenWidth, &fScreenHeight);
```

```
  // Creates background texture
  pstBitmap  = orxDisplay_CreateBitmap(256, 256);
  pstTexture = orxTexture_Create();
  orxTexture_LinkBitmap(pstTexture, pstBitmap, "BackgroundTexture");

  // Allocates pixel buffer
  au8Data = (orxU8 *)orxMemory_Allocate(256 * 256 * sizeof(orxRGBA),
orxMEMORY_TYPE_VIDEO);

  // For all pixels
  for(u32Y = 0; u32Y < 256; u32Y++)
  {
    for(u32X = 0; u32X < 256; u32X++)
    {
      orxU32  u32Index;
      orxU8   u8Value;

      // Gets pixel value
      u8Value = (orxU8)(u32X ^ u32Y);

      // Gets pixel index
      u32Index = (u32Y * 256 + u32X) * sizeof(orxRGBA);

      // Stores pixel channels
      au8Data[u32Index]     =
      au8Data[u32Index + 1] =
      au8Data[u32Index + 2] = u8Value;
      au8Data[u32Index + 3] = 0xFF;
    }
  }

  // Updates bitmap content
  orxDisplay_SetBitmapData(pstBitmap, au8Data, 256 * 256 * sizeof(orxRGBA));

  // Frees pixel buffer
  orxMemory_Free(au8Data);
```

Now here's the important bit we'd have to do even if we were to use external bitmaps for our background and spot light textures: the lightmap texture creation.

As you can see it's pretty straightforward: we create a bitmap of the same size than our display, create an empty texture and then bind them together with the name `LightMapTexture`.
That's the name we're going to use in config to reference this texture.

```
  // Creates lightmap texture
  pstBitmap   = orxDisplay_CreateBitmap(orxF2U(fScreenWidth),
orxF2U(fScreenHeight));
  pstTexture  = orxTexture_Create();
  orxTexture_LinkBitmap(pstTexture, pstBitmap, "LightMapTexture");
```

Let's now create a texture for our spotlight, a simple white circle with soft edges will do the trick. Actually all the pixels are white and we only vary the opacity (ie. the alpha value).
When a texture is rendered with an additive blend mode, it's the alpha value of a pixel that will determine how much of its color will be added to the background pixel.

```
  // Pushes main config section
  orxConfig_PushSection("Main");

  // Gets spot size & inner radius
  u32Size     = orxConfig_GetU32("SpotLightSize");
  fSpotRadius = orxConfig_GetFloat("SpotLightRadius");

  // Pops config section
  orxConfig_PopSection();

  // Creates spotlight texture
  pstBitmap   = orxDisplay_CreateBitmap(u32Size, u32Size);
  pstTexture  = orxTexture_Create();
  orxTexture_LinkBitmap(pstTexture, pstBitmap, "SpotLightTexture");

  // Allocates pixel buffer
  au8Data = (orxU8 *)orxMemory_Allocate(u32Size * u32Size * sizeof(orxRGBA),
orxMEMORY_TYPE_VIDEO);

  // For all pixels
  for(u32Y = 0; u32Y < u32Size; u32Y++)
  {
    for(u32X = 0; u32X < u32Size; u32X++)
    {
      orxU32    u32Index;
      orxU8     u8Alpha;
      orxFLOAT  fDistance;

      // Gets pixel distance from center
      fDistance = orxMath_Sqrt(orxMath_Pow(orxU2F(u32X) - (orx2F(0.5f) *
orxU2F(u32Size)), orx2F(2.0)) + orxMath_Pow(orxU2F(u32Y) - (orx2F(0.5f) *
orxU2F(u32Size)), orx2F(2.0)));

      // Gets pixel alpha
      u8Alpha = (orxU8)orxF2U(orx2F(255.0f) * orxMath_SmoothStep(orx2F(0.5f)
* orxU2F(u32Size), fSpotRadius, fDistance));

      // Gets pixel index
      u32Index = (u32Y * u32Size + u32X) * sizeof(orxRGBA);

      // Stores pixel channels
      au8Data[u32Index]     =
      au8Data[u32Index + 1] =
      au8Data[u32Index + 2] = 0xFF;
      au8Data[u32Index + 3] = u8Alpha;
    }
```

```
  }

  // Updates bitmap content
  orxDisplay_SetBitmapData(pstBitmap, au8Data, u32Size * u32Size *
sizeof(orxRGBA));

  // Frees pixel buffer
  orxMemory_Free(au8Data);
}
```

Let's now have a look at the `Init()` function that will get called when orx is executed.

First we call the `InitTextures()` function, making sure all the textures are created and ready to be used before creating any graphics/objects.

```
orxSTATUS orxFASTCALL Init()
{
  orxS32    i;
  orxSTATUS eResult = orxSTATUS_SUCCESS;

  // Creates and inits textures
  InitTextures();
```

Let's now use the `Main.ViewportList` config parameter to create as many viewports as we need.

The way it's done here makes it very easy to add new viewports/cameras without having to change the code later one.
That's how we added a control viewport to show what is exactly rendered onto the `LightMapTexture` without having to change a single line of code.

```
  // Pushes main config section
  orxConfig_PushSection("Main");

  // Creates all viewports
  for(i = 0; i < orxConfig_GetListCounter("ViewportList"); i++)
  {
    orxViewport_CreateFromConfig(orxConfig_GetListString("ViewportList",
i));
  }

  // Pops config section
  orxConfig_PopSection();
```

Let's now create our scene. It contains all our objects, including a background, different colored spotlights and a lightmap object used for the final compositing process.

```
  // Creates scene
  pstScene = orxObject_CreateFromConfig("Scene");

  // Done!
```

```
  return eResult;
}
```

Our `Run()` function doesn't do much beside listening for the `Quit` and `Screenshot` inputs.

```
orxSTATUS orxFASTCALL Run()
{
  orxSTATUS eResult = orxSTATUS_SUCCESS;

  // Screenshot?
  if(orxInput_IsActive("Screenshot") && orxInput_HasNewStatus("Screenshot"))
  {
    // Captures it
    orxScreenshot_Capture();
  }
  // Quitting?
  if(orxInput_IsActive("Quit"))
  {
    // Updates result
    eResult = orxSTATUS_FAILURE;
  }

  // Done!
  return eResult;
}
```

We don't actually do any cleaning in the `Exit()` function (we're very lazy) as all our allocations/creations were made through orx and it'll be orx's pleasure to clean everything for us.


```
void orxFASTCALL Exit()
{
  // We could delete everything we created here but orx will do it for us
anyway
}
```

And finally we have a regular main function and a console-less windows one that whose sole purpose is to execute orx.

```
int main(int argc, char **argv)
{
  // Executes orx
  orx_Execute(argc, argv, Init, Run, Exit);

  // Done!
  return EXIT_SUCCESS;
}
```

Config

Here's where the magic happens. 

As we've seen, we haven't done much related to compositing in code.
All we did was creating an empty texture named `LightMapTexture` that will serve as a target for our off-screen light rendering pass.
The rest of the code was simply to create some visual assets (background & spotlight textures), viewports&cameras and a scene.

So the secret of compositing is to be found in data/compositing.ini.

First of all, we define our custom config section `Main` with some data used to create the spotlight texture and more importantly, the list of viewports we want to create.

There are two important viewports there: `MainViewport` and `LightMapViewport`.
`LightMapViewport` has a camera that will only render the spotlights to the off-screen texture `LightMapTexture`.
When that rendering pass is done, we'll then render the full scene in `MainViewport`, to the screen this time.

Finally, `ControlViewport` will get rendered to the screen. This viewport is simply to give a visual cue to what happens behind the scene but doesn't take any part in the compositing process.

NB: The viewports are rendered in the order of their creation.

In our case the viewports are rendered in this order: `LightMapViewport`, `MainViewport`, `ControlViewport`.

```
[Main]
ViewportList    = LightMapViewport # MainViewport # ControlViewport
SpotLightSize   = 64
SpotLightRadius = 12
```

Let's now have a look at their definition.

`MainViewport` is a regular rendering viewport using a standard camera, with no extra settings.
Note that the camera will only 'see' objects whose depth Z verify `-1 < Z <= 1`.

We'll then make sure only our background, lightmap object and our eventual scene objects will be in this space.
The spotlights will have a Z outside of this range so that they won't be rendered by the `MainViewport`/`MainCamera` couple.

```
[MainViewport]
Camera          = MainCamera

[MainCamera]
FrustumWidth    = @Display.ScreenWidth
```

```
FrustumHeight   = @Display.ScreenHeight
FrustumFar      = 2
Position        = (0, 0, -1)
```

Let's now have a look to our off-screen lightmap rendering.

All we have to do for that is to specify a `BackgroundColor`[1] and a `Texture`[2] for our `LightMapViewport`.

There's nothing special about `LightMapCamera` beside the fact it'll only 'see' objects whose depth Z verify `-11 < Z <= -9`.

```
[LightMapViewport]
Camera          = LightMapCamera
Texture         = LightMapTexture
BackgroundColor = (0, 0, 0)

[LightMapCamera@MainCamera]
Position        = (0, 0, -11)
```

Lastly we define `ControlViewport` to be a vignette in the top left corner of our display.
It'll use the same camera used for lightmap rendering (`LightMapCamera`) and will thus display on screen what was rendered off-screen to the `LightMapTexture` texture.

```
[ControlViewport]
Camera          = LightMapCamera
Size            = (0.25, 0.25, 1)
UseRelativeSize = true
BackgroundColor = (20, 20, 20)
```

Let's now define all our scene objects.

We begin with the scene itself that contains a `Background`, the `LightMap` proxy object and 4 `SpotLights`.

```
[Scene]
ChildList       = Background #
                  LightMap   #
                  SpotLight1 # SpotLight2 # SpotLight3 # SpotLight4
```

The `Background` object is very straightforward.

We set it as a child of the `MainCamera` so that it'll be stretched to fill the full camera frustum, be as far as possible from the camera to be rendered first and with no blending as it's a completely opaque object (which is a rendering optimization).

It also use a graphic whose texture is named `BackgroundTexture`.
What happens here is that orx will try to see if it already has a texture with that name in store (which it does as we created it programmatically).
If it weren't to find that texture, it'll then try to load it from file[3].

```
[Background]
ParentCamera    = MainCamera
Position        = (0, 0, 0.5)
Scale           = 1
Graphic         = BackgroundGraphic
Color           = (240, 240, 240)
BlendMode       = none

[BackgroundGraphic]
Pivot           = center
Texture         = BackgroundTexture
```

Now to the `LightMap` object which is the important one for the compositing process.

We define this object to also follow the camera but this time it's very close to it so that it'll get rendered last (ie. when everything else in the scene is already rendered).

It's using the `LightMapTexture` that we created in code. However we didn't fill it with any content. But every time `LightMapViewport` is rendered, the content of the texture gets updated with all the rendered spotlights.

It's also using a `multiply` blending mode which means that every pixel of the background will have its value multiplied by the matching pixel of this object upon rendering.
If the pixel of the `LightMapTexture` is white, then the background pixel will remain 'untouched' (aka lit).
If the pixel is black, on the contrary, then the background pixel will also become black (aka in darkness).
Any intermediate value will result in apparent more of less dimly lit background pixel (including colored lighting).

That's where all the compositing 'magic' takes place. Pretty easy, isn't it? 

```
[LightMap]
ParentCamera    = MainCamera
Position        = (0, 0, 0.1)
Scale           = 1
Graphic         = LightMapGraphic
BlendMode       = multiply

[LightMapGraphic]
Pivot           = center
Texture         = LightMapTexture
```

Now to the spotlights.

They're basically using the 'SpotLightTexture' we created in code, some have been colored.
They'll all be moving/rotating independently too.

Please note that they all have a `Z = -10`, which means only the `LightMapCamera` can see them.

```
[SpotLight1]
Position        = (0, 0, -10)
Scale           = (10, 3, 1)
AngularVelocity = 90
Color           = (0, 200, 0)
Graphic         = SpotLightGraphic
BlendMode       = add

[SpotLight2@SpotLight1]
Position         = (0, -150, -10)
AngularVelocity  = 45
Color            = (200, 0, 0)

[SpotLight3@SpotLight1]
Position         = (0, 150, -10)
AngularVelocity  = 180
Color            = (0, 0, 200)

[SpotLight4@SpotLight1]
Position         = (0, 0, -10)
Scale            = 5
AngularVelocity  = 0
Color            = (120, 120, 120)
FXList           = SpotLightFX

[SpotLightGraphic]
Pivot           = center
Texture         = SpotLightTexture
```

We're skipping all other config values as they are irrelevant for the compositing process and are simply defining an orxFX to get one of the spotlight move along a sinusoidal curve and some global settings such as the display size, the screenshot file prefix and format, hiding the mouse, etc...

That's all, folks! 

1) otherwise lightmap rendering will accumulate from one frame to another

2) the famous `LightMapTexture` we created in code

3) but we have no file named `BackgroundTexture` in the same folder as our executable so that would fail