Papervision3D clouded planet Earth tutorial and source - Code walkthrough

by Benny Bottema - Monday, June 15, 2009

http://www.bennybottema.com/papervision3d-clouded-planet-earth-tutorial-and-source-code-walkthrough/

This page describes the code used in the Papervision3D clouded planet Earth tutorial and source post.

Planets and Earth

Since I'm trying to learn Papervision3D to make a spaceshooter, I've decided to make a generic Planet class and a subclass, Earth, which provides specific planet details such as textures and clouds. This way I can easily create new kinds of planets, including ones with procedurally generated textures.

In pseudo code, here's how the Planet and Earth classes work together:

```
class Planet extends DisplayObject3D {
 constructor {
  createAddAndConfigureSpheresAndPlanes(createMaterials());
  configureLayeringAndMasking();
 }

 abstract function createMaterials() {
  // provided by subclasses (Earth)
 }
}

class Earth extends Planet {
 override function createMaterials() {
  // create Earth materials
 }
}
```

So in effect I've separated how the planets are created from how the materials are created, which greatly reduces the boilerplate code we'd need and gives us a proper place to keep our embedded textures.

Planet code

Let's begin with the Planet baseclass.

- Planet.as

```
public function Planet(viewport:Viewport3D, diameter:Number, sunLight:
PointLight3D = null, quality:Number = QUALITY_MEDIUM,
    planetRotationSpeed:Number = .3, cloudsRotationSpeed:Number = .15,
 updateFrequency:Number = 125, phasePerSecond:Number = 1) {
```

As you can see the constructor is rather large. This is because there are many aspects of the planet's quality you can control. In addition there are some parameters that go to Planet's superclass RealtimeDisplayObject3D. I haven't mentioned this class yet, because it is outside of the scope of this article and I've written a blogpost about it earlier. In short, it is a very simple class that performs updates based on time events rather than frame events, so that we can update a planet in realtime (such as the clouds) instead of how many fps we get.

The quality parameter only controls the quality of the main sphere with the land texture. In theory it could be passed down to the subclasses in the createMaterial() methods giving the subclasses like Earth a chance to change the quality of the materials. This way you could for example create three versions of the same planet for different distances from the camera, so that far away planets have very low detail quality.

```
// for some reason I can't get the planes for circle masks and fresnel
l
// glow to get the exact same diameter as the spheres
const GLOWSIZE:Number = diameter * 1.06;
const GRADIENT_QUALITY:Number = 500;

addChild(_planet = new Sphere(createPlanetMaterial(), diameter / 2, qu
ality, quality));
addChild(clouds = new Sphere(createCloudMaterial(), diameter / 2, QUAL
ITY_LOW, QUALITY_LOW));
addChild(planetMask = new Plane(createPlanetMaskMaterial(GRADIENT_QUAL
ITY), GLOWSIZE, GLOWSIZE));
addChild(cloudMask = new Plane(createPlanetMaskMaterial(GRADIENT_QUALI
TY), GLOWSIZE, GLOWSIZE));
addChild(glow = new Plane(createGlowMaterial(GRADIENT_QUALITY, [0, .1,
 .3, 0], [0xBB, 0xE9, 0xFA, 0xFF]), GLOWSIZE, GLOWSIZE));
```

The glowsize constant determines both the size of the masking discs and the planet's glow. These need to be the same size since otherwise you could get a glow outside the planet. Also, I've been unable to determine the right size for the planes with the gradient masking discs to exactly cover the planet, so I've used a magic modifier of 1.06.

As you can see the sphere's are created with materials obtained from other methods like createPlanetMaterial(). These are provided by the Earth subclass.

```
// apparently planes always need to be flipped over (or have doublesid
ed materials)
glow.geometry.flipFaces();
planetMask.geometry.flipFaces();
cloudMask.geometry.flipFaces();
```

For these planes to work they should always face the camera, or otherwise the planet's fresnell glow for example would completely mess up. For this I use *planetMask.lookAt(camera)*, which works fine except that they face the opposite direction. For some reason the lookAt() function makes the planes look at the opposite direction.

To solve this you can make the material doublesided, or you can just flip the normals (faces) of the planes.

```
// structure layers so that the glow is always on top of the planet
var containerLayer:ViewportLayer = PV3DUtil.createLayer(viewport, null
, viewport.containerSprite);
var planetLayer:ViewportLayer = PV3DUtil.createLayer(viewport, planet,
 containerLayer, 0);
var cloudLayer:ViewportLayer = PV3DUtil.createLayer(viewport, clouds,
containerLayer, 1);
var planetMaskLayer:ViewportLayer = PV3DUtil.createLayer(viewport, pla
netMask, containerLayer, 2);
var cloudMaskLayer:ViewportLayer = PV3DUtil.createLayer(viewport, clou
dMask, containerLayer, 3);
var glowLayer:ViewportLayer = PV3DUtil.createLayer(viewport, glow, con
tainerLayer, 4);
containerLayer.sortMode = ViewportLayerSortMode.INDEX_SORT;

/**
 * (Utility function in a separate utility class, PV3DUtil)
 * Creates a preconfigured ViewportLayer. Preconfigured are: layerInde
x, parent layer, added DisplayObject3D.
 */
public static function PV3DUtil.createLayer(viewport:Viewport3D, do3d:
DisplayObject3D = null, parent:ViewportLayer = null, layerIndex:Number
 = 0):ViewportLayer {
 var viewportLayer:ViewportLayer = new ViewportLayer(viewport, do3d);
 viewportLayer.layerIndex = layerIndex;
 if (parent != null) {
```

```
  parent.addLayer(viewportLayer);
 }
 return viewportLayer;
}
```

I've used layers for everything here, because now I can manage exactly what is shown on top of what. This is necessary since the default sorting mode is Z-based; In this scenario it can happen that the glow is rendered beneath the spheres. To avoid a lot of boilerplate code I've made a utility function out of this in a class I made called PV3DUtil.

```
// apply planet mask, so that the planet is guaranteed covered by the
glow
planetLayer.cacheAsBitmap = planetMaskLayer.cacheAsBitmap = true;
cloudLayer.cacheAsBitmap = cloudMaskLayer.cacheAsBitmap = true;
planetLayer.mask = planetMaskLayer;
cloudLayer.mask = cloudMaskLayer;
```

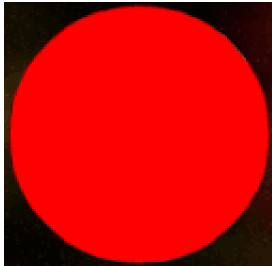
This is a tricky bit I had to use Google a lot for. In Flash, alpha masks only work if **both the masked sprite and the masking sprite are cached as bitmaps**. Ultimately, everything in Papervision3D is drawn to ordinary bitmap objects. Luckily the Papervision3D folks realized this and added many opportunities to make use of this: in our case by masking spheres by planes through the use layers.

```
/**
 * Defines a masking disc the same size as the fresnell glow 'around'
Earth.
 */
private function createPlanetMaskMaterial(size:Number):MaterialObject3
D {
 var planetMaskTexture:Sprite = PV3DUtil.createGradientSprite(size, [0
, 0], [1, 0], [0xFA, 0xFF]);
 return new MovieMaterial(planetMaskTexture, true);
}

/**
 * (Utility function in a separate utility class, PV3DUtil)
 * Performs a basic gradient fill and returns it on a new sprite of th
e specified size. Used to avoid boilerplate code.
 */
public static function PV3DUtil.createGradientSprite(size:Number, colo
rs:Array, alphas:Array, ratios:Array):Sprite {
```

```
 var mat:Matrix = new Matrix();
 mat.createGradientBox(size, size);
 var sprite:Sprite = new Sprite();
 sprite.graphics.beginGradientFill(GradientType.RADIAL, colors, alphas
, ratios, mat);
 sprite.graphics.drawRect(0, 0, size, size);
 sprite.graphics.endFill();
 return sprite;
}
```

This method creates the gradient masking disc, which is opaque until the very end. There is a fading border on the edge of about 1 or 2 pixels thick to take of the edge of the mask.



So this mask is applied to the entire planet, both land/cloud spheres. The planet's glow goes on top of all this which has the exact same size as the mask so that it fits nicely.

```
/**
 * Rotates Earth and clouds separately. Makes sure the glow- and mask
planes are pointed towards the camera
 */
public override function update(camera:Camera3D):void {
 planet.yaw(planetRotationSpeed);
 clouds.yaw(cloudsRotationSpeed);
 glow.lookAt(camera);
 planetMask.copyTransform(glow);
 cloudMask.copyTransform(glow);
 super.update(camera);
}
```

This should be straightforward. Rotate the planet a little bit as well as the clouds. Make the glow plane look at the camera and have the masking planes do the same by simply copying this information from the glow plane.

One note is in order perhaps: I've mentioned Planet extends RealtimeDisplayObject3D. I haven't used this to use realtime rotation for the planets, which I could've done as well. Instead, rotation is done based on frame events while Earth updates the clouds animation (not rotation) based on time events.

Earth code

```
[Embed (source="../../../../assets/earth.jpg")]
private var BitmapEarth:Class;
[Embed (source="../../../../assets/clouds2.png")]
private var BitmapClouds:Class;
[Embed (source="../../../../assets/earth_heightmap.jpg")]
private var BitmapHeightmap:Class;
```

One of the neat things of having subclasses for planets like this is the fact that you can now store textures per planet in a separate class. To me that's nice and clean code.

```
private var realtimeCloudsTexture:RealtimeCloudsTexture;
```

This is the magical realtime cloud object. We'll store a reference to it in Earth, so we can update these clouds as we see fit. This *realtimeCloudsTexture* is actually a glorified Sprite which is used in a shaded material.

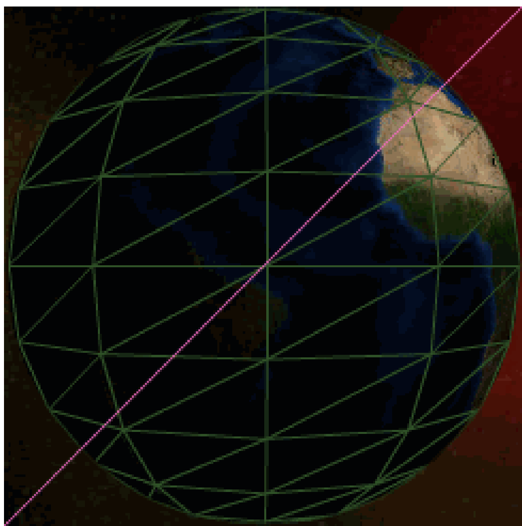
```
public function Earth(viewport:Viewport3D, diameter:Number, sunLight:P
ointLight3D = null) {
 super(viewport, diameter, sunLight, Planet.QUALITY_MEDIUM, .3, .15, 1
00, 5);
}
```

The last four parameters are planet rotation speed, cloud rotation speeds, realtime update frequency in millimeter and phase value per second spread out over the updates per seconds (the frequency).

```
/**
 * Defines the geological texture as a phong shaded material.
 */
protected override function createPlanetMaterial():MaterialObject3D {
 var earthBitmap:BitmapData = new BitmapEarth().bitmapData;
 var heightmapBitmap:BitmapData = new BitmapHeightmap().bitmapData;
```

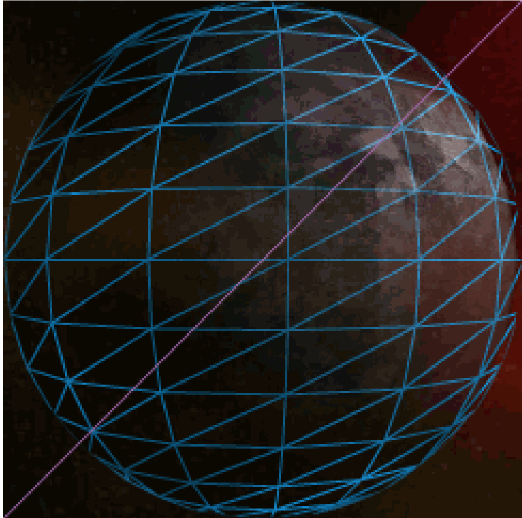
```
 var earthShader:PhongShader = new PhongShader(sunLight, 0xFFFFFF, 0x1
11111, 20, heightmapBitmap, null);
 return new ShadedMaterial(new BitmapMaterial(earthBitmap), earthShade
r);
}
```

This is where it gets interesting. Here, we're creating a PhongShader using the geological data texture, so that we can apply a sunlight to it. In addition we're using a heightmap as bumpmap to make the planet's surface a little bit more dynamic and interesting.



```
/**
 * Defines the clouds texture as a phong shaded material. The clouds t
exture
 * contains a runtime-
maintained alpha channel so you can look through the animated clouds.
 */
protected override function createCloudMaterial():MaterialObject3D {
 realtimeCloudsTexture = new RealtimeCloudsTexture(new BitmapClouds().
bitmapData);
 var realtimeCloudsMaterial:MovieMaterial = new MovieMaterial(realtime
CloudsTexture, true, true);
 var cloudShader:PhongShader = new PhongShader(sunLight, 0xFFFFFF, 0x0
00000, 20, null, null);
 return new ShadedMaterial(realtimeCloudsMaterial, cloudShader);
}
```

The cloud material is exactly like the geological data material, except instead of passing a BitmapMaterial into the ShadedMaterial, we're passing in a MovieMaterial. This is because now we can specify that our texture has transparency and that it is animated. Transparency because we still want to see the Earth underneath the clouds and animated because we're providing realtime clouds updates.



Cloud code

As stated in the blog post, the clouds have an alpha channel which is updated in runtime with a perlin noise map.

```
public class RealtimeCloudsTexture extends Sprite {
 // original clouds texture to perform realtime visiblity mapping on
 private var cloudsBitmap:BitmapData;

 private var phase1:Number = 0; // used for octave 1 in perlin noise m
ap
 private var phase2:Number = 0; // used for octave 2 in perlin noise m
ap
 private var phase3:Number = 0; // used for octave 3 in perlin noise m
ap

 public function RealtimeCloudsTexture(cloudsBitmap:BitmapData) {
  this.cloudsBitmap = cloudsBitmap;
  // for some reason we need to prepare sprite size in the constructor
, so let's just do that
  PV3DUtil.drawBitmapToSprite(new BitmapData(cloudsBitmap.width, cloud
sBitmap.height), this);
 }
}
```

So the phases represent the progression of the clouds in time and map directly to the octaves properties in the perlin noise map we're using. The cloudsBitmap passed in is never modified. Instead we treat it like a blueprint, copying it each frame and applying the alpha channel modifications. This was necessary to avoid stacked alpha channel modifications.

```
/**
 * Updates the clouds phase (visibility map) using perlin noise.
 *
 * 1). Perlin Noise is used for alpha channel assigned to the clouds t
exture
 * 2). Since we're drawing perlin noise in runtime, let's just keep th
e perlin size very small
 * and just resize the noise to the target bitmap.
 */
public function update(stepsize:Number):void {
 // produce phased perlin noise alpha map
 var point1:Point = new Point(phase1 -= stepsize * 2);
 var point2:Point = new Point(phase2 += stepsize);
 var point3:Point = new Point(phase3 -= stepsize);
 var cloudsPhase:BitmapData = new BitmapData(120, 120);
 // pick any channel as long the same channel is used to convert to al
pha channel when combing it all
 cloudsPhase.perlinNoise(20, 20, 2, 754, true, true, BitmapDataChannel
.RED, false, [point1, point2]);

 // resize alpha map to fit the cloudstexture
 var cloudsPhaseResized:BitmapData = new BitmapData(cloudsBitmap.width
, cloudsBitmap.height);
 var m:Matrix = new Matrix();
 m.scale(cloudsBitmap.width / cloudsPhase.width, cloudsBitmap.height /
 cloudsPhase.height);
 // draw resized visibility map and slightly increase contrast to comp
letely hide/show clouds
 cloudsPhaseResized.draw(cloudsPhase, m, new ColorTransform(1, 1, 1, 1
, -25, -25, -25));

 // combine everything on a new bitmap
 var cloudsCombined:BitmapData = new BitmapData(cloudsBitmap.width, cl
oudsBitmap.height);
 cloudsCombined.copyPixels(cloudsBitmap, cloudsBitmap.rect, new Point(
0, 0));
 cloudsCombined.copyChannel(cloudsPhaseResized, cloudsBitmap.rect, new
 Point(0, 0), BitmapDataChannel.RED, BitmapDataChannel.ALPHA);

 graphics.clear();
 PV3DUtil.drawBitmapToSprite(cloudsCombined, this);
```

```
}
```

So this is actual code that modifies the cloud coverage. Four steps happen here:

1. Create a low quality progressed perlin noise map
2. Resize the perlin noise map to the size of the clouds texture
3. Apply the resized noise map to a copy of the clouds texture blueprint
4. Draw the entire combination on itself as the new clouds sprite used in the moviematerial

---

PDF generated by Kalin's PDF Creation Station