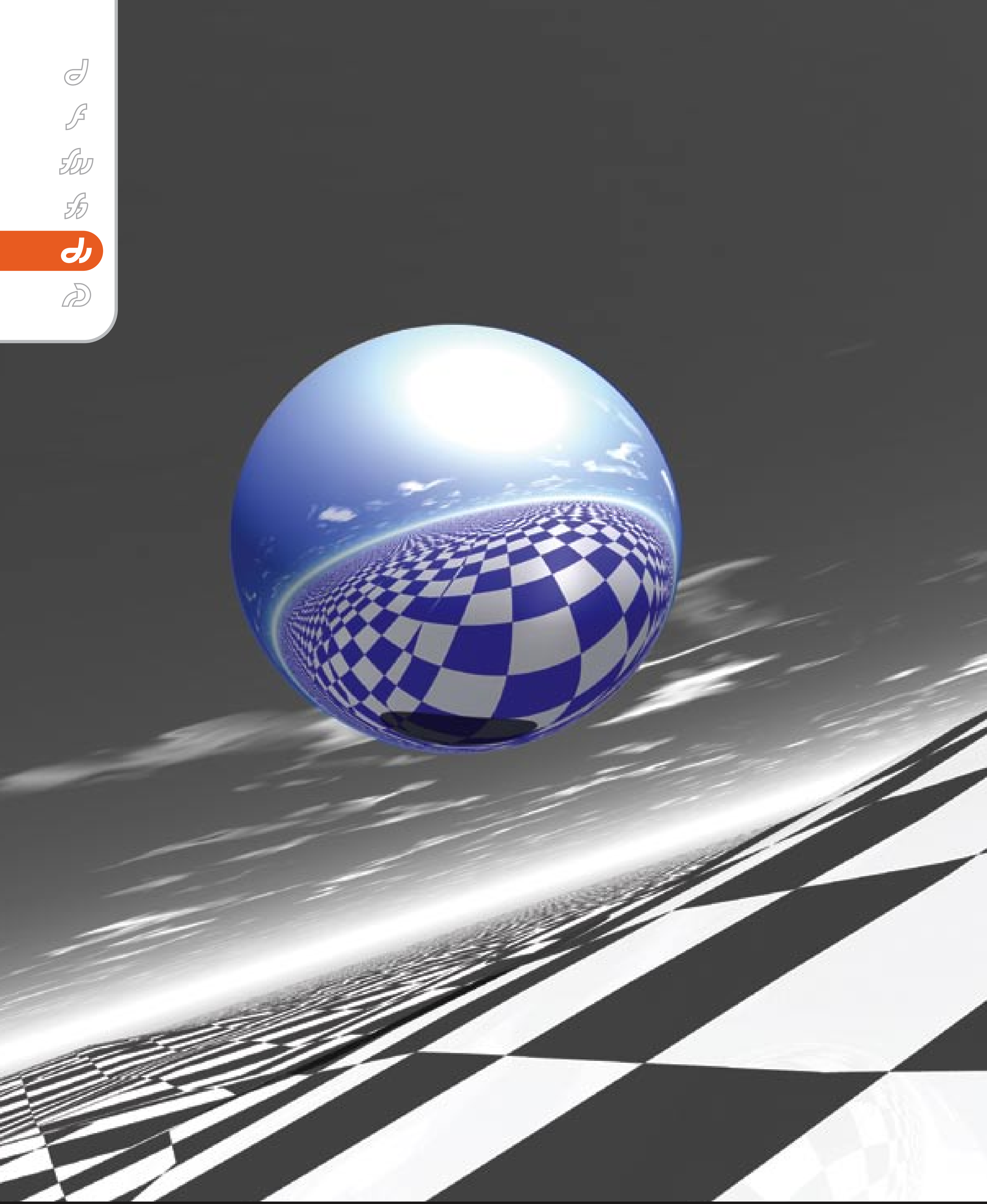
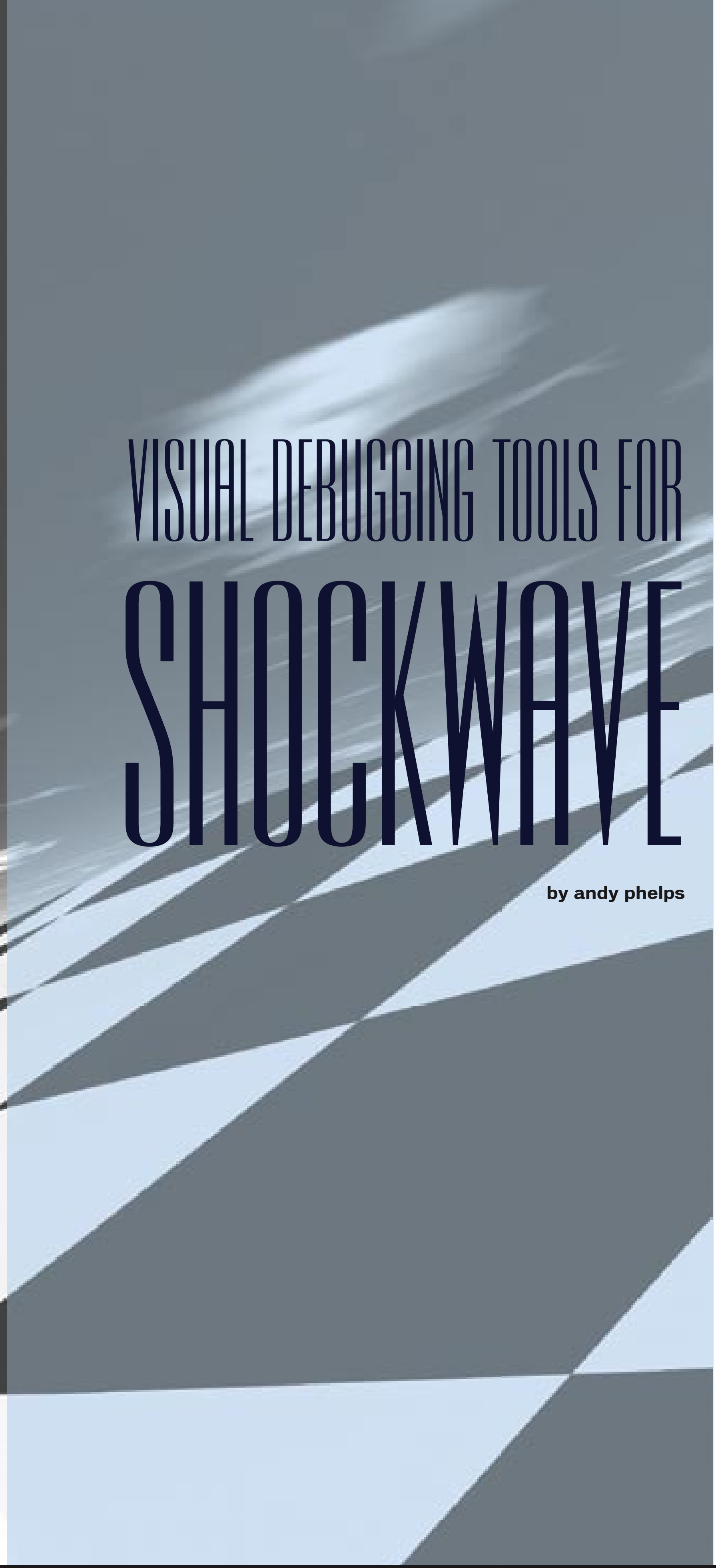




VISUAL DEBUGGING TOOLS FOR SHOCKWAVE 3D

by andy phelps

Recently, I have been working in Shockwave3D in Lingo and JavaScript syntax. Regardless of what I am writing, I keep running into issues that can be summed up by the following phrase: I don't know exactly what I'm doing. More precisely, I don't know exactly what I am doing when I am doing it. I'm figuring it out. And that's what programming interactive things is all about: finding a way to do what you need to do, such that the user has the experience you want them to have. Unfortunately it is sometimes difficult to tell exactly what your code is doing, and this is particularly true in 3D environments.

So with these issues in the back of my head, I was working on my lightmap generation tool, and I was really getting stuck. A "lightmapper" is a tool that pre-generates, through raytracing or other means, a series of maps that represent the lighting in a 3D scene as textures on the individual objects. This is a very popular technique in game-level design, as Brian Robbins noted at MAX, and as several authors have noted in the game development community. Figure 1 shows a few sample renders from my tool in progress.

The problem I was having was in projecting the shadows, and in particular figuring out the angle to each light from the points along the surface. I was never really "sure" of exactly where the ray was I was checking against. This is relatively easy to figure out for a single light, but I was getting very confused when calculating multiple light sources (see Figure 2). In order to figure out just what was going on, I used a strategy that I have used in the past" build a visual "prop," or "stand-in" of the ray itself. The only problem is that Shockwave3D has no #line primitive. No problem! Just make a very long, skinny triangle. Listing 1 shows a Lingo handler that creates a "connector" object: it has a triangle that it uses as a "line," which it can snap between a beginning and end point. Additionally, it can color each end of the "line" a different color, and blends the two together along the triangle face.

Using these lines, I was able to trace out each and every light path in the scene, and get a sense of whether or not it was doing what I wanted it to do (see Figure 3). This was very handy, but this tool isn't limited to that specific use. I've used it in the past to represent surface normals (something I wish S3D had a #debug flag for), direction vectors, a "point at" vector between two objects, and even rotational axes when I haven't been able to see the ones drawn through the #debug flag. (For some reason, the axes generated by Director are all black when drawn with the DirectX7_0 renderer on my nVidia cards). Because each end of the "line" can be colored separately, they can be used to represent directional vectors, rather than just a straight connection between two points.

In order to get the "lines" to show up well in a debug environment, I generally create a custom shader that ramps up the emissive and ambient qualities (and sets some flags so that the vertex colors actually have an effect). For the script in Listing 1, I used the shader props set in Listing 2. I've also used "lines" in a similar way in JavaScript syntax, a very similar script to the Lingo ThreeDLine implementation is presented in Listing 3.

Being able to "see" what is actually going on has proved invaluable for this and several other projects. Using color and shape makes it much easier to tell what the code is doing than an iterative 'run and see' approach. By thinking of debugging visually, tools can be constructed that make it easier to see just what is going on. If you have either already built tools, or are thinking of writing some after seeing these simple examples, I encourage you to share them with the community as you are able, to build up a library of visual aids that help us in our daily work.

figure 1

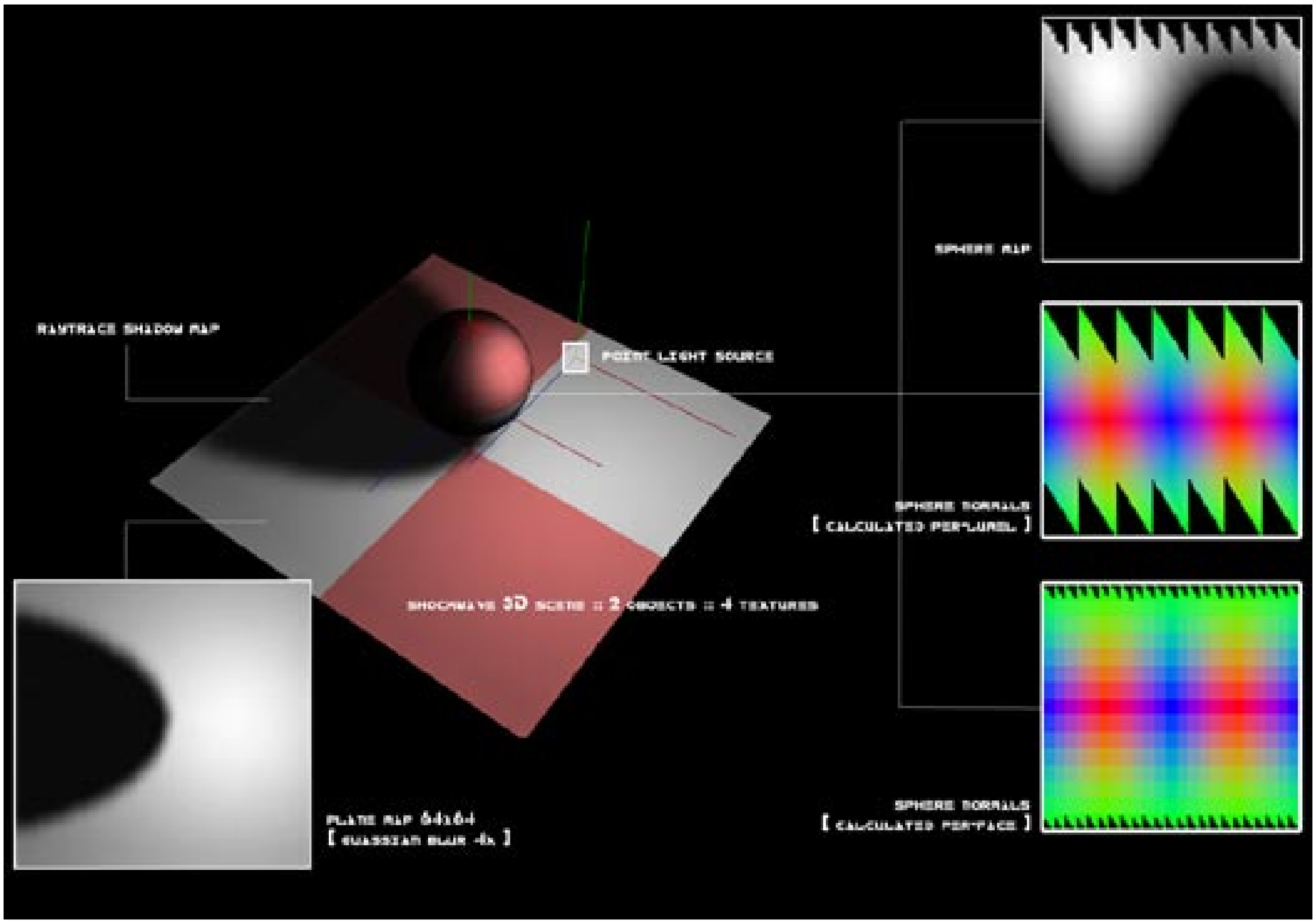


figure 2

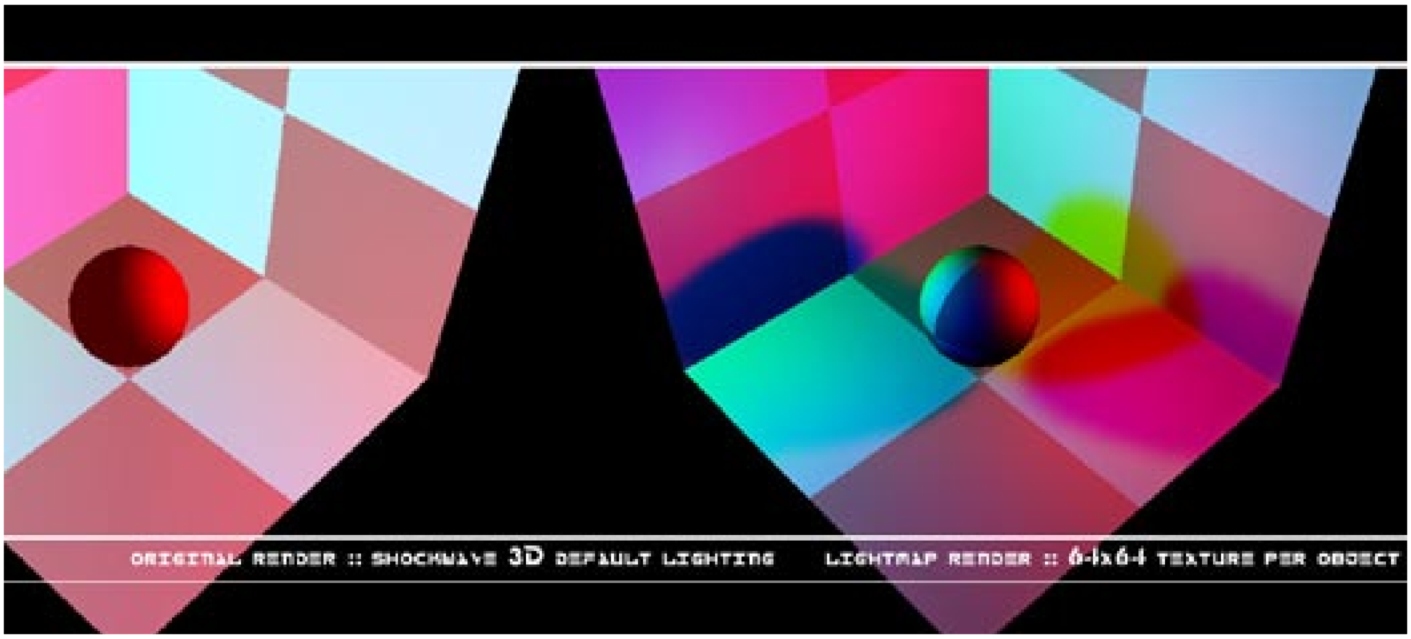
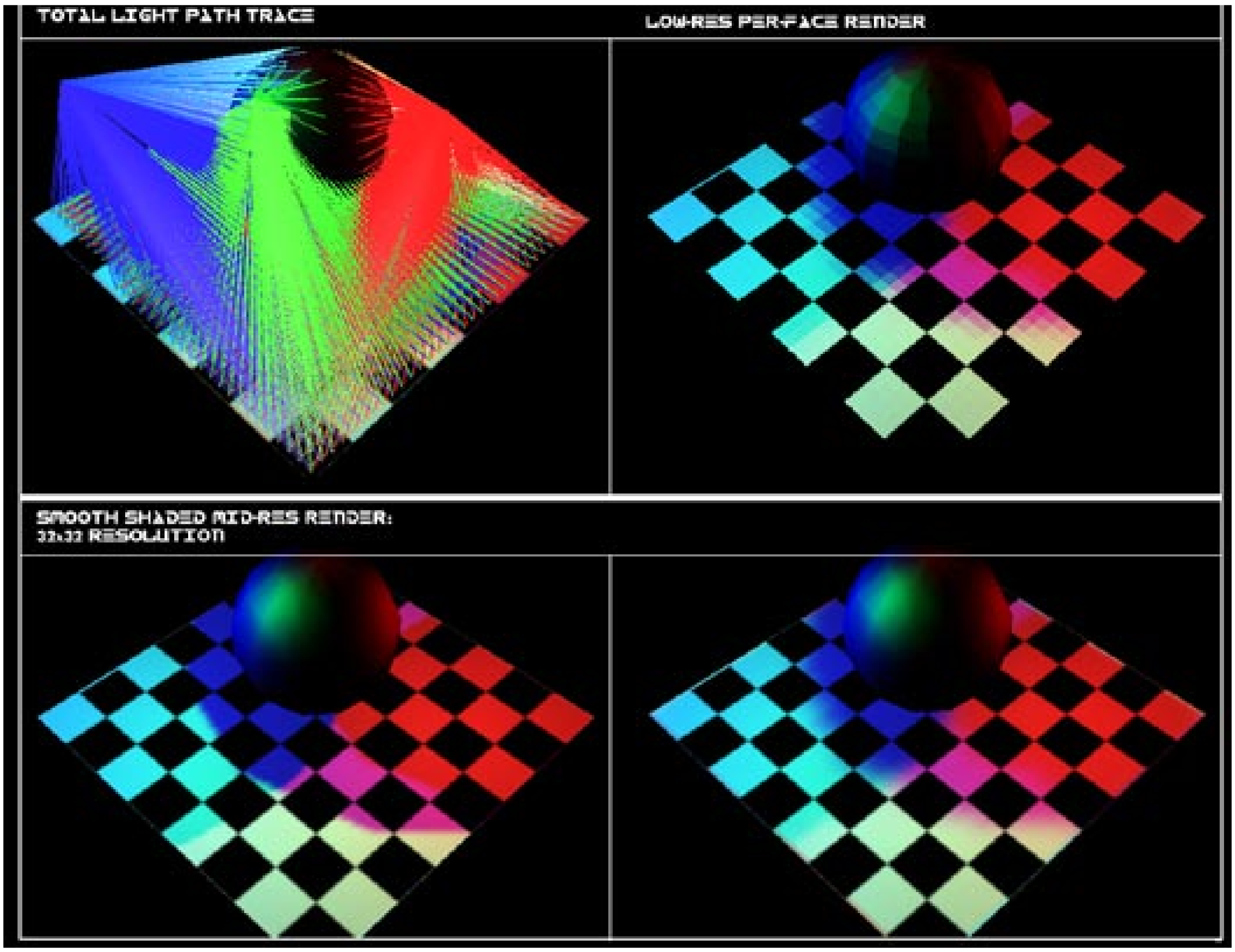

ORIGINAL RENDER :: SHOCKWAVE 3D DEFAULT LIGHTING
LIGHTMAP RENDER :: 64x64 TEXTURE PER OBJECT

figure 3



listing 1

```
-- ThreeDLine
-----------------------------------------------------
--------
property p_vPosA     -- position A of line
property p_vPosB     -- position B of line
property p_mshMesh  --line mesh
property p_mModel   --line model

-----------------------------------------------------
--------
-- ThreeDLine::New()
-- a_sName      - string name of the line to be created
-- a_aColor     - array of 2 rgb colors [start, end]
-- a_vPoint1    - vector start of line
-- a_vPoint2    - vector end of line
-- a_shShader   - shader to be applied to the line
------------------------------------------------------
--------
on new me, a_sName, a_aColor, a_vPoint1, a_vPoint2,
a_shShader

  me.p_vPosA = a_vPoint1
  me.p_vPosB = a_vPoint2

  --create a mesh for this connector
  if (voidP(_global.D3D_WORLD[#g_3DWorld].model(a_
sName))) then
  else
    return 0
  end if

  me.p_mshMesh = _global.D3D_WORLD[#g_3DWorld].newMesh(
\
                                           a_
sName,1,3,3,3,0)

  --set color list
  me.p_mshMesh.colorList  = [ a_aColor[1], \
                             a_aColor[2], \
                             a_aColor[1] ]

  --set vertex list
  me.p_mshMesh.vertexList = [ vector(0,0,0), \
                             vector(0,100,0), \
                             vector(100,100,0) ]

  --set normal list
  --NOTE: this is a hack, normals here make ok
  --lines, but are not technically correct for 3D
  --lighting...
  me.p_mshMesh.normalList = [ vector(1,1,1), \
                             vector(1,1,1), \
                             vector(1,1,1) ]

  --set the vertices and colors into the mesh
  --NOTE: Need to use getPropRef to parse the lingo
arrays
  --that are embedded in the S3D Xtra
  me.p_mshMesh.face[1].vertices = [1,2,3]
  me.p_mshMesh.face[1].colors = list(1,2,3)

  --build the triangle
  me.p_mshMesh.build()

  --create a model from our triangle
  me.p_mModel = _global.D3D_WORLD[#g_3DWorld].newModel(
\
                                       a_sName, me.p_
mshMesh)


  --set render ops for best debug lines
  me.p_mModel.visibility = #both

  --set shader on this model
  me.p_mModel.shader = a_shShader

  me.mUpdate()

  return me
end ThreeDLine
------------------------------------------------------
--------



------------------------------------------------------
--------
-- ThreeDLine::mUpdate()
-- a_vPointA  - vector new start position
-- a_vPointB  - vector new end position
------------------------------------------------------
--------
on mUpdate me, a_vPointA, a_vPointB
  -- make the connector position itself between the A
and B
  -- parent nodes.  This is a little but tricky.
  me.p_vPosA = a_vPointA
  me.p_vPosB = a_vPointB

  vPosC = vector( me.p_vPosA.x, \
                 me.p_vPosA.y - 2.000, \
                 me.p_vPosA.z - 2.000)
  me.p_mshMesh.vertexList = [me.p_vPosA, me.p_vPosB,
vPosC]

end mUpdate
```

listing 2

```
----------------------------------------------------
-------------[ set shader properties for  ]----------
-------------[ debug lines: glow and wire ]----------
lineShader.shininess = 0
lineShader.blend = 10
lineShader.transparent = true
lineShader.blendFunction = #blend
lineShader.blendConstant = 30
lineShader.texture = void
lineShader.specular = color(0,0,0)
lineShader.diffuse  = color(0,0,0)
lineShader.ambient  = color(0,0,0)
lineShader.emissive = color(255,255,255)
lineShader.renderStyle  = #wire
```

Andrew Phelps is an assistant professor in the B. Thomas Golisano College of Computing and Information Sciences at the Rochester Institute of Technology. He has an academic background in information technology, as well as traditional fine arts and computer animation. His work using Director has been featured at the Director-Online User's Group (DOUG) as well as the DevNet Center at Macromedia. Andy regularly teaches coursework in multimedia programming, game programming, and simulation/visualization.

amp@it.rit.edu

```
lineShader.flat = false
lineShader.useDiffuseWithTexture = false
-------------[ end shader properties ]---------------
```

listing 3

```
/*----------------------------------------------------
---------
// Connector::Connector a line that connects 2 nodes,
A & B
// a_oNodeA node objectA
// a_oNodeB node objectB
------------------------------------------------------
--------*/
function Connector(a_oNodeA, a_oNodeB) {

  //set props
  this.p_oNodeA = a_oNodeA;
  this.p_oNodeB = a_oNodeB;
  this.p_bIsTerminator = a_bIsTerminator;
  this.p_sName = "CONNECTOR:_" + a_oNodeA.mGetName() +
                              "_" + a_oNodeB.mGetName();

  //create a mesh for this connector
  this.p_mshMesh = _global.gD3D_WORLD.newMesh(this.
p_sName,

 1,3,3,3,0);


  //set color list
  this.p_mshMesh.colorList  = list( this.p_oNodeA.mGet-
Color(),
                                    this.p_oNodeB.
mGetColor(),
                                    this.p_oNodeA.
mGetColor());

  //set vertex list
  this.p_mshMesh.vertexList = list( vector(0,0,0),
                                    vector(0,100,0),
                                    vec-
tor(100,100,0));

  //set normal list
  //NOTE: this is a hack, normals here make ok
  //lines, but are not technically correct for 3D
  //lighting...
  this.p_mshMesh.normalList = list( vector(0,1,0),
                                    vector(0,1,0),
                                    vector(0,1,0));

  //set the vertices and colors into the mesh
  //NOTE: Need to use getPropRef to parse the lingo
arrays
  //that are embedded in the S3D Xtra
  this.p_oFace = this.p_mshMesh.getPropRef("face", 1)
  this.p_oFace.vertices = list(1,2,3);
  this.p_oFace.getPropRef("face", 1).colors =
list(1,2,3);

  //build the triangle
  //this.p_mshMesh.generateNormals(symbol("flat"));
  this.p_mshMesh.build();

  //create a model from our triangle
  this.p_mModel = _global.gD3D_WORLD.newModel(this.
p_sName,

 this.p_mshMesh);

  //set visibility of the triangle to two-sided
  this.p_mModel.visibility = symbol("both");

  //assign custom shader, reset properties as needed
  this.p_mModel.shader = _global.gD3D_WORLD.
getProp("shader",2);
  this.p_mModel.shader.shininess = 0;
  this.p_mModel.shader.blend = 10;
  this.p_mModel.shader.emissive = color(50,50,50);

  //call initial update to position the connecting tri-
angle
  this.mUpdate();
}


/*----------------------------------------------------
---------
//Connector::mGetName
//get the name of this connector as a string
------------------------------------------------------
-------*/
Connector.prototype.mGetName = function() {
  return this.p_sName
}


/*----------------------------------------------------
---------
//Connector::mUpdate
//position line between nodes A & B, call after moving
//either A, B, or both.
------------------------------------------------------
-------*/
Connector.prototype.mUpdate = function() {


  //make the connector position itself between the A
and B
  //parent nodes.  This is a little but tricky.

  var vPosA = this.p_oNodeA.mGetPosition();
  var vPosB = this.p_oNodeB.mGetPosition();

  var vPosC = vector( vPosA.x,
                      vPosA.y - 2.000,
                      vPosA.z - 2.000);
  this.p_mshMesh.vertexList = list(vPosA, vPosB,
vPosC);
}
```