Implementation of a CML Boiling Simulation using Graphics Hardware

Mark J. Harris
University of North Carolina at Chapel Hill

1 Introduction

This report, which supports the paper [Harris, et al. 2002], serves to elucidate many of the details of implementing the coupled map lattice (CML) boiling simulation of [Yanagita 1992].

2 Overview

Yanagita's boiling simulation models the evolution of temperature in a liquid as it is heated by a bottom plate. The CML simulation consists of four suboperations that model components of the behavior: boundary conditions, diffusion, thermal buoyancy, and latent heat (see Figure 1). The rest of this report will detail each of the four suboperations of boiling. For an overview of implementing CML simulations using graphics hardware, see [Harris, et al. 2002].

Our implementation runs in OpenGL using NVIDIA specific extensions available on the NVIDIA GeForce 3 and GeForce 4. To ease programming the GPU, we use a parser library from NVIDIA called NVPARSE. For details on the usage of NVPARSE, see [Spitzer 2001;Wynn 2001]. Note that the code samples that follow are for demonstration only. In our actual implementation, the code is much more modular.

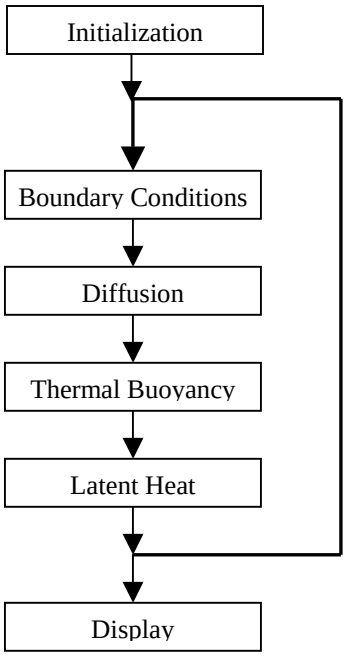

Figure 1: Operation of a CML boiling simulation.

3 Initialization

The initialization phase of this simulation allocates textures needed for storage of the state of the simulation, including intermediate state created by each pass. Listing 1 gives code for this phase. The function InitSimulation() creates one texture for each pass (there are seven passes per iteration of the boiling simulation). It then calls an initialization function for each of the four operations of the boiling simulation. Each of these initialization functions creates a display list for each pass of the operation it is initializing. Each display lists sets the register combiners and texture shader state for the operation, binds input textures to the appropriate texture units, renders a quadrilateral using the correct texture coordinate offsets for the operation, and then copies the results from the frame buffer into the output texture for the operation. After initializing the simulation operations, InitSimulation() calls ResetSimulation(), which simply resets the input texture of the simulation to a uniform gray value modulated with random noise. The noise helps add randomness to the simulation as it proceeds.

3.1 Boundary Condition Application

Listing 3 gives code to initialize the boundary condition operation. This operation simply takes the input to the simulation (which may be the results of the previous iteration), and sets the top and bottom edges to fixed temperatures. It does so by blending a precomputed boundary texture (transparent everywhere except the top and bottom rows of texels) with its input texture. It then copies the results of this blending to its own output texture. The function InitBoundaries() stores the OpenGL calls that perform this functionality in a display list.

3.2 Thermal Diffusion

Listing 4 gives code to initialize the diffusion operation. This operation implements diffusion via the equation

$$T'_{i,j} = T_{i,j} + \frac{c_d}{4}\nabla^2 T_{i,j},$$

as described in Appendix A of [Harris, et al. 2002]. When it is applied, temperature is diffused through the lattice. The register combiners code used to average the texture samples from the four neighbors of each node can be seen in Listing 3. NVPARSE is used to convert this code into OpenGL calls, which are stored in a display list with the rest of the OpenGL calls for the diffusion operation.

3.3 Thermal Buoyancy

Listing 5 gives code to initialize the thermal buoyancy operation. This is a two-pass operation on GeForce 3 and 4, because it needs to sample three textures, and perform table lookups using the values of two of the samples. The table lookups are performed using dependent texturing. Therefore, this operation would need five texture units to operate in a single pass. The function InitBuoyancy() creates a display list that stores the texture shader and register combiners OpenGL calls, along with other calls to perform the above operation and copy the results into an output texture.

The thermal buoyancy operation causes regions of temperature at or above the boiling point to float upward due to the difference in density between liquid and gas. It implements the equation

$$T'_{i,j} = T_{i,j} - \tfrac{\sigma}{2} T_{i,j}[\rho(T_{i,j+1}) - \rho(T_{i,j-1})],$$

$$\rho(T) = \tanh[\alpha(T - T_c)],$$

as described in [Harris, et al. 2002]. The tanh function is performed via the "DEPENDENT_GB_TEXTURE_2D_NV" texture shader operation as described above. The result texture from the diffusion pass is used as input (T) to a dependent texture lookup into a precomputed texture that contains values of $\rho(T)$ for the fixed T_c, which is the boiling point temperature. This lookup is performed twice, once for a texture sample at the node above the current node in the diffusion texture, and once for a sample at the node below.

3.4 Latent Heat Transfer Initialization

Listing 6 gives code to initialize the latent heat transfer operation. When boiling liquid changes into gas, it absorbs a small amount of latent heat from its surroundings. When gas cools and changes into liquid, it liberates a small amount of latent heat into its surroundings. The latent heat operator implements this by evaluating the following [Yanagita 1992]:

$$\begin{aligned} &\text{if } T^b_{i,j} > T_c \text{ and } T^t_{i,j} < T_c \text{ then} \\ &\quad T^{t+1}_{n(i,j)} = T^b_{n(i,j)} - \eta \\ &\text{if } T^b_{i,j} < T_c \text{ and } T^t_{i,j} > T_c \text{ then} \\ &\quad T^{t+1}_{n(i,j)} = T^b_{n(i,j)} + \eta \end{aligned}$$

Here $T^b_{i,j}$ and $T^b_{n(i,j)}$ are the value of node (i,j) and each of its four nearest neighbors after the buoyancy operation. $T^t_{i,j}$ is the value of node (i,j) after the previous iteration of the boiling simulation, and $T^{t+1}_{n(i,j)}$ is the value of each of the four nearest neighbors of node (i,j) that will be written as output of the current latent heat transfer operation. T_c is the boiling point temperature, and η is the latent heat of phase change. The result is that if a node changed from liquid to gas in the current iteration, a small amount of temperature, η is subtracted from each of its neighbor node. If the node changed from gas to liquid, η is added to each of its neighbor nodes.

The problem with this operation is that it is inherently a *scatter* operation, since when it is applied to a node (i, j), the operation must modify the neigbors of (i, j). Programmable texture blending cannot operate this way. It performs only *gather* operations, and can only modify the value of the node it is currently processing. Luckily, we can invert this operation. The inverted operation looks like this:

$$\begin{aligned} &T^{t+1}_{i,j} = T^b_{i,j} \\ &\text{foreach neighbor } n(i, j) \\ &\quad \text{if } T^b_{n(i,j)} > T_c \text{ and } T^t_{n(i,j)} < T_c \text{ then} \\ &\quad\quad T^{t+1}_{i,j} = T^{t+1}_{i,j} + \eta \\ &\quad \text{else if } T^b_{n(i,j)} < T_c \text{ and } T^t_{n(i,j)} > T_c \text{ then} \\ &\quad\quad T^{t+1}_{i,j} = T^{t+1}_{i,j} - \eta \end{aligned}$$

When applied in a SIMD fashion to all nodes of the lattice, this operator accomplishes the same function as the previous one.

To implement this simple looking code in NVIDIA register combiners is somewhat difficult. It requires 2 texture samples for each of four neighbors, plus a texture sample at node (i,j) itself, meaning it requires three passes. Register combiners provide only a very primitive form of conditional statements in the form of a multiplex (mux) operation with no branching. Because the mux can only be performed on a specific component of a specific register, getting values into the correct format and location within the register combiners requires a lot of code. As you can see from Listing 6, the first two passes of this operation use all eight of the general combiners available on GeForce 3 and 4.

The function InitLatentHeat creates two display lists that store the register combiners and texture shader code that implements the behavior described above, along with other OpenGL state and calls to evaluate the first two passes of the latent heat operator. It creates another display list for the third pass of the operation, which combines the results of the first two passes and renders the combination to the frame buffer. The result is then copied to the output texture. Since this is the last pass in an iteration of the boiling simulation, this is the texture that is displayed to the user.

4 Execution and Display

Since all of the OpenGL calls and state needed to execute the seven passes of the boiling simulation are stored in display lists,

a single iteration of the simulation can be executed simply by calling the display lists in order using glCallLists(). Listing 2 gives the function SimulationStep() which does this. This function also sets the correct view port and view frustum to setup a one-to-one mapping from pixels to texels.

Displaying the simulation is easy. Simply bind the output texture of the final pass of the simulation, and use it to texture a quadrilateral (or any other object) displayed to the user. Listing 2 includes the function DisplayOutput() which does this.

5 References

[Harris, et al. 2002] Harris, M.J., Coombe, G., Scheuermann, T. and Lastra, A. Physically-Based Visual Simulation on Graphics Hardware. *Submitted to Graphics Hardware 2002*. 2002.

[Spitzer 2001] Spitzer, J. *Programmable Texture Blending*. 2001.

[Wynn 2001] Wynn, C. *Texture Shader Configuration using NVPARSE*. 2001.

[Yanagita 1992] Yanagita, T. Phenomenology of boiling: A coupled map lattice model. *Chaos*, *2* 3. 343-350. 1992.

6 Code Listings

The following code listings are taken from a working demonstration of our graphics hardware CML boiling simulation. The complete code is available at http://www.cs.unc.edu/~harrism/cml/dl/SimpleCMLBoil.zip .

6.1 Listing 1: Simulation Initialization

```
//------------------------------------------------------------------------
// Function    : InitSimulation
// Description : Initialize all display lists and state used by the sim.
//------------------------------------------------------------------------
void InitSimulation(unsigned int width, unsigned int height)
{
  iWidth = width;
  iHeight = height;

  if (!glh_init_extensions("GL_NV_texture_shader "
                           "GL_NV_register_combiners "
                           "GL_NV_register_combiners2 "
                           "GL_ARB_multitexture"))
  {
    printf("The application failed to initialize the following required "
           "OpenGL extension(s): \n %s\n",
           glh_get_unsupported_extensions());
    return false;
  }

  unsigned int i;
  glGenTextures(PASS_COUNT, iOutputTextureIDs);

  for (i = 0; i < PASS_COUNT; ++i)
  {
    glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[i]);

    // Just allocate it, so we can do fast copies later
    // with glCopyTexSubImage2D().
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, iWidth, iHeight, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, NULL);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
  }

  rPerTexelWidth  = 1.0f / (float)iWidth;
  rPerTexelHeight = 1.0f / (float)iHeight;

  InitBoundaries(rBottomT, rTopT);
  InitDiffusion(rDiffusion);
  InitBuoyancy(rAlpha, rSigma, rTc);
  InitLatentHeat(rTc, rEta);

  ResetSimulation();
}

//------------------------------------------------------------------------
// Function    : ResetSimulation
// Description : Restart the simulation.
//------------------------------------------------------------------------
void ResetSimulation()
```

```
{
  unsigned int i;

  // load a noisy initial field to make the simulation interesting.
  unsigned char *pData = new unsigned char[iWidth * iHeight];
  float rNF = 0.2f / 32768.0f;
  for (i = 0; i < iWidth * iHeight; ++i)
  {
    float rVal = (rNF * rand() - 0.1f) + 0.5f * (rBottomT + rTopT);
    pData[i] = (unsigned char) (255 * rVal);
  }
  glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_INPUT]);
  glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, iWidth, iHeight,
                  GL_LUMINANCE, GL_UNSIGNED_BYTE, pData);

  delete [] pData;
}
```

6.2 Listing 2: Simulation Execution and Display

```
//------------------------------------------------------------------------
// Function             : SimulationStep
// Description     : Execute a single step of the simulation.
//------------------------------------------------------------------------
void SimulationStep()
{
  // to correctly update the lattice, we must make pixels map 1:1 to texels.
  // this requires an orthographic frustum with corners fit to the range
  // [-1,1], and a viewport of the same resolution as the lattice.
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(-1, 1, -1, 1);
  glClearColor(0, 0, 0, 0);
  glViewport(0, 0, iWidth, iHeight);

  // Simple: just call each of the display lists in order!
  glCallLists(PASS_COUNT, GL_UNSIGNED_INT, iStateDisplayListIDs);

  glDisable(GL_REGISTER_COMBINERS_NV);
  glDisable(GL_TEXTURE_SHADER_NV);
}

//------------------------------------------------------------------------
// Function    : DisplayOutput
// Description : Display the result of a simulation step.
//------------------------------------------------------------------------
void DisplayOutput()
{
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(-1.1, 1.1, -1.1, 1.1);
  glClearColor(0.2, 0.2, 0.2, 0.2);
  glViewport(0, 0, glutGet(GLUT_WINDOW_WIDTH), glutGet(GLUT_WINDOW_HEIGHT));
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glActiveTextureARB(GL_TEXTURE0_ARB);
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_OUTPUT]);

  RenderQuad(PASS_DISPLAY);
}
```

6.3 Listing 3: Boundary Condition Initialization

```
//------------------------------------------------------------------------
// Function    : InitBoundaries
// Description : Initialize the pass that applies fixed boundary
//                    conditions at the top and bottom of the lattice.  The
//                    bottom is the heat plate (1 pass).
//------------------------------------------------------------------------
```

```
void InitBoundaries(float iBottomTemp, float iTopTemp)
{
  unsigned int i, k;
  unsigned char *pBoundaryData = new unsigned char[iWidth * iHeight * 2];

  // create the data for the fixed boundaries. initialize the whole array to 0
  memset(pBoundaryData, 0, sizeof(unsigned char) * iWidth * iHeight * 2);

  k = 2 * (iWidth * (iHeight-1)); // top left
  for (i = 0; i < iWidth * 2; i += 2, k += 2)
  { // set the temperature at the top and bottom edges.
    // Set the bottom row to the temperature of the heated plate.
    pBoundaryData[i+0] = (unsigned int) (255 * iBottomTemp);
    pBoundaryData[i+1] = 255; // opaque (rest of texture is transparent)

    // Set the top temperature to something cooler, since temp. radiates here.
    pBoundaryData[k+0] = (unsigned int) (255 * iTopTemp);
    pBoundaryData[k+1] = 255; // opaque (rest of texture is transparent)
  }

  // create the fixed boundary conditions texture.
  glBindTexture(GL_TEXTURE_2D, iBoundaryTextureID);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, iWidth, iHeight, 0,
               GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE, pBoundaryData);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
  delete [] pBoundaryData;

  // create the display list for the boundary conditions pass.
  iStateDisplayListIDs[PASS_BOUNDARIES] = glGenLists(1);

  // this display list encapsulates the state and operation of the
  // fixed boundary condition application pass.
  glNewList(iStateDisplayListIDs[PASS_BOUNDARIES], GL_COMPILE);
  {
    // This call to nvparse sets texture unit 0 to do a standard 2D texture op
    nvparse("!!TS1.0      \n"
            "texture_2d();\n"
            "nop();       \n"
            "nop();       \n"
            "nop();       \n");
    glEnable(GL_TEXTURE_SHADER_NV);
    glDisable(GL_REGISTER_COMBINERS_NV); // No register combiners for this op.

    // first render the output of the previous pass into the buffer to be
    // blended with the boundaries.
    glActiveTextureARB(GL_TEXTURE0_ARB);
    glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_INPUT]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

    RenderQuad(PASS_BOUNDARIES);

    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_BLEND);

    glActiveTextureARB(GL_TEXTURE0_ARB);
    glBindTexture(GL_TEXTURE_2D, iBoundaryTextureID);

    RenderQuad(PASS_BOUNDARIES);

    glDisable(GL_BLEND);

    // Now copy the resulting pixels into the output texture
    glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_BOUNDARIES]);
    glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, iWidth, iHeight);
  }
  glEndList();
}
```

6.4 Listing 4: Diffusion Initialization

```
//-----------------------------------------------------------------------------
// Function    : InitDiffusion
// Description : Initialize the pass that performs temperature diffusion
//               (1 pass).
//-----------------------------------------------------------------------------
void InitDiffusion(float rDiffusionCoefficient)
{
  int i;
  // Create the state display list for the diffusion pass.
  iStateDisplayListIDs[PASS_DIFFUSION] = glGenLists(1);

  // This display list encapsulates the state and operation of the
  // diffusion pass.
  glNewList(iStateDisplayListIDs[PASS_DIFFUSION], GL_COMPILE);
  {
    // This call to nvparse sets up the register combiners to
    // average the four inputs.
    nvparse("!!RC1.0                          \n"
            "{                                \n"
            "  rgb                            \n"
            "  {                              \n"
            "    discard = half_bias(tex0);   \n"
            "    discard = half_bias(tex1);   \n"
            "    spare0 = sum();              \n"
            "  }                              \n"
            "}                                \n"
            "{                                \n"
            "  rgb                            \n"
            "  {                              \n"
            "    discard = half_bias(tex2);   \n"
            "    discard = half_bias(tex3);   \n"
            "    spare1 = sum();              \n"
            "  }                              \n"
            "}                                \n"
            "{                                \n"
            "  rgb                            \n"
            "  {                              \n"
            "    discard = spare0;            \n"
            "    discard = spare1;            \n"
            "    spare0 = sum();              \n"
            "    scale_by_one_half();         \n"
            "  }                              \n"
            "}                                \n"
            "{                                \n"
            "  rgb                            \n"
            "  {                              \n"
            "    discard = spare0;            \n"
            "    discard = unsigned_invert(zero); \n"
            "    spare0 = sum();              \n"
            "    scale_by_one_half();         \n"
            "  }                              \n"
            "}                                \n"
            "out.rgb = spare0;                \n"
            "out.a = unsigned_invert(zero); \n");

    glEnable(GL_REGISTER_COMBINERS_NV);

    // nvparse texture shader
    // This call to nvparse sets all texture units to do a standard
    // 2D texture operation.
    nvparse("!!TS1.0\n"
            "texture_2d();\n"
            "texture_2d();\n"
            "texture_2d();\n"
            "texture_2d();\n");
    glEnable(GL_TEXTURE_SHADER_NV);

    // bind the input texture to all four texture units.
    for (i = 0; i < 4; i++)
    {
      glActiveTextureARB(GL_TEXTURE0_ARB + i);
      glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_BOUNDARIES]);
      glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

```
      glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
      // linear filtering is required to sample 4 nearest neighbors AND
      // the center texel (filtering will compute a weighted avg. of each
      // center-neighbor pair, the result of which will be passed as
      // inputs to the register combiners, where they will be averaged.
      glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
      glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    }

    // render a screen quad. with texture coords doing offset by
    // diffusion coefficient.
    RenderQuad(PASS_DIFFUSION, rDiffusionCoefficient);

    // Now we need to copy the resulting pixels into the output texture
    glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_DIFFUSION]);
    glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, iWidth, iHeight);
  }
  glEndList();
}
```

6.5 Listing 5: Thermal Buoyancy Initialization

```
//------------------------------------------------------------------------
// Function    : InitBuoyancy
// Description : Initialize the operation that computes and applies
//               thermal buoyancy (2 passes).
//------------------------------------------------------------------------
void InitBuoyancy(float rAlpha, float rSigma, float rPhaseChangeTemp)
{
  // Create the phase change lookup table for the buoyancy operator.
  float rStepSize = 1.0f / 255.0f;
  unsigned char *pData = new unsigned char[256 * 4];

  int k = 0;
  for (int i = 0; i < 256; i++)
  {
    float rValue = tanh(20 * (rAlpha * (rPhaseChangeTemp - rStepSize * i)));
    // scale and bias value from [-1,1] range to [0,1] range
    rValue = 255 * (0.5f + rValue * 0.5f);
    pData[k++] = (unsigned char)rValue;
    pData[k++] = (unsigned char)rValue;
    pData[k++] = (unsigned char)rValue;
    pData[k++] = 255;
  }

  // allocate the phase change lookup table texture.
  glGenTextures(1, &iPhaseChangeTextureID);
  glBindTexture(GL_TEXTURE_2D, iPhaseChangeTextureID);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 256, 1, 0,
               GL_RGBA, GL_UNSIGNED_BYTE, pData);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

  delete [] pData;

  ////////////////////////////////////////////////////////////////////////
  // state setup for first pass
  ////////////////////////////////////////////////////////////////////////

  // create the state display list for the first buoyancy pass.
  iStateDisplayListIDs[PASS_BUOYANCY_1] = glGenLists(1);

  glNewList(iStateDisplayListIDs[PASS_BUOYANCY_1], GL_COMPILE);
  {
    // this nvparse code sets up the texture shaders to use the "temperature"
    // read from two samples of a texture bound to units 0 and 1 as input to
    // the function rho(T) = tanh(alpha*(Tc - T), which is implemented as a
    // dependent texture read into a 1D lookup table texture.
    nvparse("!!TS1.0\n"
            "texture_2d();\n"
            "texture_2d();\n"
            "dependent_gb(tex0);\n"
            "dependent_gb(tex1);\n");
```

```
  glEnable(GL_TEXTURE_SHADER_NV);

  // The nvparse call below parses a register combiners program that
  // takes 2 samples lying on either side (above and below) the center texel.
  // They should be the result of a lookup into a
  // rho(T) = tanh(alpha*(Tc - T)) texture, and the values should be between
  // 0 to 1, but represent signed [-1,1] values, so they
  // need to be expanded first.
  //
  // Gets colors from 2 texture stages (The other 2 are used as input to do
  // dependent GB texturing.):
  // tex2 = bottom side:  rho(T(x,y-1))  // i.e. set texcoord to (x,y-1)
  // tex3 = top side:  rho(T(x,y+1))  // i.e. set texcoord to (x,y+1)
  //
  // result is: T' = sigma * (tex2 - tex3) = sigma * (rho(tex0) - rho(tex1));
  // [ scaled and biased to [0,1] so that 0->0.5 ]
  nvparse("!!RC1.0                                    \n"
          "{                                          \n"
          "  //placeholder for sigma                 \n"
          "  const1 = (0.64, 0.64, 0.64, 1);         \n"
          "  rgb                                      \n"
          "  {                                        \n"
          "    discard = expand(tex2) * const1;       \n"
          "    discard = -expand(tex3) * const1;      \n"
          "    spare0 = sum();                        \n"
          "  }                                        \n"
          "}                                          \n"
          "{                                          \n"
          "  rgb                                      \n"
          "  {                                        \n"
               "    // bias up to [0, 1] to store.        \n"
               "    discard = spare0;                     \n"
               "    discard = unsigned_invert(zero);      \n"
               "    spare0 = sum();                       \n"
          "    scale_by_one_half();                   \n"
          "  }                                        \n"
          "}                                          \n"
          "out.rgb = spare0;                          \n"
          "out.a = unsigned_invert(zero);             \n");

  glEnable(GL_REGISTER_COMBINERS_NV);

  //  pass sigma to register combiner stage 0, constant 0.
  float pSigma[4] = { rSigma * 0.5f,
                      rSigma * 0.5f,
                      rSigma * 0.5f,
                      rSigma * 0.5f };
  glCombinerStageParameterfvNV(GL_COMBINER0_NV, GL_CONSTANT_COLOR1_NV, pSigma);

  // bind the output texture of the diffusion pass to units 0 and 1,
  // and the phase change lookup table texture to units 2 and 3.
  for (i = 0; i < 4; ++i)
  {
    glActiveTextureARB(GL_TEXTURE0_ARB + i);
    glBindTexture(GL_TEXTURE_2D,
                  i < 2 ?
                  iOutputTextureIDs[PASS_DIFFUSION] :
                  iPhaseChangeTextureID);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  }

  RenderQuad(PASS_BUOYANCY_1);

  // Now we need to copy the resulting pixels into the intermediate texture
  glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_BUOYANCY_1]);

  // use CopyTexSubImage for speed (even though we copy all of it) since we
  // pre-allocated the texture
  glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, iWidth, iHeight);
}
glEndList();
```

```
  ////////////////////////////////////////////////////////////////////////
  // state setup for second pass
  ////////////////////////////////////////////////////////////////////////

  // create the state display list for the second buoyancy pass.
  iStateDisplayListIDs[PASS_BUOYANCY_2] = glGenLists(1);

  glNewList(iStateDisplayListIDs[PASS_BUOYANCY_2], GL_COMPILE);
  {
    // The nvparse call below parses a register combiners program that
    // takes 2 texture samples -- both at the center texel location, but in
    // different textures.  The first texture is the current
    // temperature field (same as input to the first buoyancy pass).
    // the second texture sample is the result of the first pass.
    // The second sample is subtracted from the temperature field sample.
    //
    // The combination of the tanh dependent GB texture lookup of the first
    // pass with this pass effectively computes the following:
    //
    // T"(x,y) = T'(x,y) - sigma/2 * T'(x,y) * [rho(T'(x,y-1)) - rho(T'(x,y+1))],
    // where rho(T) = tanh(alpha*(Tc - T))
    // and Tc is the phase transition threshold temperature.
    nvparse("!!RC1.0                                    \n"
            "{                                          \n"
            "  rgb                                      \n"
            "  {                                        \n"
            "    discard = tex0;                        \n"
            "    discard = -expand(tex1) * tex0;        \n"
            "    spare0 = sum();                        \n"
            "  }                                        \n"
            "}                                          \n"
            "out.rgb = spare0;                          \n"
            "out.a = unsigned_invert(zero);              \n");

    // this nvparse code sets up the texture shaders to do a normal
    // 2D texture lookup on each of the first two texture units, in
    // order to pass the current temperature state and the result of
    // the first pass to the register combiners program above.
    nvparse("!!TS1.0\n"
            "texture_2d();\n"
            "texture_2d();\n"
            "nop();\n"
            "nop();\n");

    glEnable(GL_TEXTURE_SHADER_NV);
    glEnable(GL_REGISTER_COMBINERS_NV);

    glActiveTextureARB(GL_TEXTURE0_ARB);
    glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_DIFFUSION]);
    glActiveTextureARB(GL_TEXTURE1_ARB);
    glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_BUOYANCY_1]);

    RenderQuad(PASS_BUOYANCY_2);

    // Now we need to copy the resulting pixels into the intermediate force
    // field texture
    glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_BUOYANCY_2]);
    glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, iWidth, iHeight);
  }
  glEndList();
}
```

6.6 Listing 6: Latent Heat Transfer Initialization

```
//----------------------------------------------------------------------------
// Function    : InitLatentHeat
// Description : Initialize the latent heat transfer operation (3 passes).
//----------------------------------------------------------------------------
void InitLatentHeat(float rPhaseChangeTemp, float rEta)
{
  // create the state display list for the first latent heat pass.
  iStateDisplayListIDs[PASS_LATENT_HEAT_1] = glGenLists(1);

  // A nearest neighbor Latent Heat Effect CML operation.
```

```
//
// This register combiners script takes 2 pairs of samples
// at the center of texels on either side (either vertically,
// or horizontally) of the center texel.  The pairs are a
// neighbor sample in both the previous pass temperature field,
// and the temperature field after thermal diffusion and bouyancy
// operators are applied.
//
// This effectively computes the following:
//
// For each of the two neighbors, n:
// {
//   If T'n(x, y) > Tc and Ttn(x, y) < Tc then output(x, y) += eta
//   If T'n(x, y) < Tc and Ttn(x, y) > Tc then output(x, y) -= eta
// }
//
// By following two passes (one horizontal, one vertical) of this operation
// with a final latent heat pass, which adds the results of the first two
// passes to T",the results of the thermal buoyancy step, we compute this:
//
// For each of the 4 neighbors, n:
// {
//   If T'n(x, y) > Tc and Ttn(x, y) < Tc then Tt+1(x, y) = T"(x, y) + eta
//   If T'n(x, y) < Tc and Ttn(x, y) > Tc then Tt+1(x, y) = T"(x, y) - eta
// }
//
// This script uses samples from all 4 texture stages:
// tex0 = First  Neighbor Tt(x, y) // i.e. set coord to address of neighbor
// tex1 = First  Neighbor T"(x, y) // i.e. set coord to address of neighbor
// tex2 = Second Neighbor Tt(x, y) // i.e. set coord to address of neighbor
// tex3 = Second Neighbor T"(x, y) // i.e. set coord to address of neighbor
//
// Tc is the phase change temperature.
// global const1 = (Tc, Tc, Tc, 1);
//    or default (0.5, 0.5, 0.5, 1) will be used!
//
// Eta is the coefficient of latent heat (rate of absorption or liberation).
// combiner2 and 5 const1 = (Eta, Eta, Eta, 1);
//    or default (0.01, 0.01, 0.01, 1) will be used!
//
// result is half of:
//
// Tt+1(x, y) = T"(x, y);
// For each nearest neighbor n
// {
//   a = Tnt - Tc;
//   b = a * (Tn" - Tc);
//   a = 0.5 + 0.5 * a;     // scale and bias from [-1, 1] to [0, 1]
//   b = 0.5 * 0.5 * b;     // scale and bias from [-1, 1] to [0, 1]
//   c = (a < 0.5) ? eta : -eta;
//   d = (b < 0.5) ? 1 : 0;
//   Tt+1(x, y) += c * d;
// }
char pLHScript[] =
  "!!RC1.0       \n"
  "const0 = (0.5, 0.5, 0.5, 0.5); \n"
  "const1 = (0.5, 0.5, 0.5, 1);   \n"
  "{ // gc0                       \n"
  "  rgb                          \n"
  "  {                            \n"
  "    discard = spare0; // dummy \n"
  "    discard = spare0;          \n"
  "    discard = sum();           \n"
  "  }                            \n"
  "  alpha                        \n"
  "  {                            \n"
  "    discard = tex0.b;          \n"
  "    discard = -const0.b;       \n"
  "    spare0.a = sum(); // a = (T'' - Tc) \n"
  "  }                            \n"
  "}                              \n"
  "{ // gc1                       \n"
  "  rgb                          \n"
  "  {                            \n"
```

```
"    discard = spare0.a * tex1; \n"
"    discard = -spare0.a * const0; \n"
"    spare1 = sum(); // b = (T'' - Tc) * (Tt - Tc) \n"
"  }                              \n"
"  alpha                          \n"
"  {                              \n"
"    discard = spare0.a;          \n"
"    discard = unsigned_invert(zero); \n"
"    spare0.a = sum();            \n"
"    scale_by_one_half();         \n"
"    // a = 0.5 * [1 + (T'' - Tc)]   [put in [0,1] range for mux] \n"
"  }                              \n"
"}                                \n"
"{ // gc2                        \n"
"  const1 = (0.01, 0.01, 0.01, 1); \n"
"  rgb                            \n"
"  {                              \n"
"    discard = -const1;           \n"
"    discard = const1;            \n"
"    tex0 = mux(); // a = (a < 0.5) ? -eta : +eta \n"
"  }                              \n"
"  alpha                          \n"
"  {                              \n"
"  discard = spare1.b;            \n"
"  discard = unsigned_invert(zero); \n"
"  spare0.a = sum();              \n"
"  scale_by_one_half();           \n"
"  // b = 0.5 * [1 + (T''-Tc)*(Tt - Tc)] [put in [0,1] range for mux] \n"
"  }                              \n"
"}                                \n"
"{ // gc3                        \n"
"  rgb                            \n"
"  {                              \n"
"    discard = tex0;              \n"
"    discard = zero;              \n"
"    tex1 = mux();                \n"
"    // b = (b < 0.5) ? a : 0     \n"
"  }                              \n"
"  alpha // start the next sample \n"
"  {                              \n"
"    discard = tex2.b;            \n"
"    discard = -const0.b;         \n"
"    spare0.a = sum(); // c = (T'' - Tc) \n"
"  }                              \n"
"}                                \n"
"{ // gc4                        \n"
"  rgb                            \n"
"  {                              \n"
"    discard = spare0.a * tex3; \n"
"    discard = -spare0.a * const0; \n"
"    spare1 = sum(); // d = (T'' - Tc) * (Tt - Tc) \n"
"  }                              \n"
"  alpha                          \n"
"  {                              \n"
"    discard = spare0.a;          \n"
"    discard = unsigned_invert(zero); \n"
"    spare0.a = sum();            \n"
"    scale_by_one_half();         \n"
"    // c = 0.5 * [1 + (T'' - Tc)] [put in [0,1] range for mux] \n"
"  }                              \n"
"}                                \n"
"{ // gc5                        \n"
"  const1 = (0.01, 0.01, 0.01, 1); \n"
"  rgb                            \n"
"  {                              \n"
"    discard = -const1;           \n"
"    discard = const1;            \n"
"    tex2 = mux();                \n"
"    // c = (c < 0.5) ? -eta : +eta \n"
"  }                              \n"
"  alpha                          \n"
"  {                              \n"
"    discard = spare1.b;          \n"
"    discard = unsigned_invert(zero); \n"
```

```
  "    spare0.a = sum();          \n"
  "    scale_by_one_half();       \n"
  "  }                            \n"
  "}                              \n"
  "{ // gc6                       \n"
  "  rgb                          \n"
  "  {                            \n"
  "    discard = tex2;            \n"
  "    discard = zero;            \n"
  "    tex3 = mux();              \n"
  "    // d = (d < 0.5) ? c : 0   \n"
  "  }                            \n"
  "  alpha                        \n"
  "  {                            \n"
  "    discard = tex1.b;          \n"
  "    discard = unsigned_invert(zero); \n"
  "    spare0.a = sum(); // b = b + 1 \n"
  "  }                            \n"
  "}                              \n"
  "{ // gc7                       \n"
  "  rgb                          \n"
  "  {                            \n"
  "    discard = spare0.a;        \n"
  "    discard = tex3;            \n"
  "    spare0.rgb = sum();        \n"
  "    // b = b + d + 1           \n"
  "    scale_by_one_half();       \n"
  "  }                            \n"
  "}                              \n"
  "out.rgb = spare0; // out = 0.5 * [b + d + 1] \n"
  "out.a = unsigned_invert(zero); \n";

////////////////////////////////////////////////////////////////////////
// state setup for first pass
////////////////////////////////////////////////////////////////////////

// pass 1
glNewList(iStateDisplayListIDs[PASS_LATENT_HEAT_1], GL_COMPILE);
{
  int i;
  nvparse(pLHScript);

  // for combiner parameters
  float pTc[4]  = { rPhaseChangeTemp, rPhaseChangeTemp, rPhaseChangeTemp, 1};
  float pEta[4] = { rEta, rEta, rEta, 1};

  glCombinerStageParameterfvNV(GL_COMBINER0_NV, GL_CONSTANT_COLOR0_NV, pTc);
  glCombinerStageParameterfvNV(GL_COMBINER1_NV, GL_CONSTANT_COLOR0_NV, pTc);
  glCombinerStageParameterfvNV(GL_COMBINER3_NV, GL_CONSTANT_COLOR0_NV, pTc);
  glCombinerStageParameterfvNV(GL_COMBINER4_NV, GL_CONSTANT_COLOR0_NV, pTc);
  glCombinerStageParameterfvNV(GL_COMBINER2_NV, GL_CONSTANT_COLOR1_NV, pEta);
  glCombinerStageParameterfvNV(GL_COMBINER5_NV, GL_CONSTANT_COLOR1_NV, pEta);

  // texture shader
  nvparse("!!TS1.0\n"
          "texture_2d();\n"
          "texture_2d();\n"
          "texture_2d();\n"
          "texture_2d();\n");

  glEnable(GL_TEXTURE_SHADER_NV);
  glEnable(GL_REGISTER_COMBINERS_NV);

  for (i = 0; i < 4; ++i)
  {
    glActiveTextureARB(GL_TEXTURE0_ARB + i);
    glBindTexture(GL_TEXTURE_2D,
                  (i % 2) ?
                    iOutputTextureIDs[PASS_INPUT] :
                    iOutputTextureIDs[PASS_BUOYANCY_2]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

```
  }

  RenderQuad(PASS_LATENT_HEAT_1);

  // Now we need to copy the resulting pixels into the intermediate texture
  glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_LATENT_HEAT_1]);
  glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, iWidth, iHeight);
}
glEndList();

iStateDisplayListIDs[PASS_LATENT_HEAT_2] = glGenLists(1);

////////////////////////////////////////////////////////////////////////
// state setup for second pass
////////////////////////////////////////////////////////////////////////
glNewList(iStateDisplayListIDs[PASS_LATENT_HEAT_2], GL_COMPILE);
{
  // texture shaders and register combiner settings
  // remain in effect from previous pass (we're just sampling the other two
  // neighors.

  for (int i = 0; i < 4; ++i)
  {
    glActiveTextureARB(GL_TEXTURE0_ARB + i);
    glBindTexture(GL_TEXTURE_2D,
                  (i % 2) ?
                    iOutputTextureIDs[PASS_INPUT] :
                    iOutputTextureIDs[PASS_BUOYANCY_2]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  }

  RenderQuad(PASS_LATENT_HEAT_2);

  // Now we need to copy the resulting pixels into the intermediate texture
  glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_LATENT_HEAT_2]);
  glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, iWidth, iHeight);
}
glEndList();

iStateDisplayListIDs[PASS_LATENT_HEAT_3] = glGenLists(1);

////////////////////////////////////////////////////////////////////////
// state setup for third pass
////////////////////////////////////////////////////////////////////////
glNewList(iStateDisplayListIDs[PASS_LATENT_HEAT_3], GL_COMPILE);
{
  // This register combiners script takes a sample in the temperature
  // field after thermal diffusion and bouyancy operators are applied,
  // along with two samples from from the previous two passes of the
  // latent heat operator.  (one in the horizontal and one in the
  // vertical dimension).  It averages the two samples, then does a
  // signed addition of them, and scales and biases the result into
  // the [0,1] range.
  //
  // For each of the 4 neighbors, n
  // If T'n(x, y) > Tc and Ttn(x, y) < Tc then Tt+1_n(x, y) = T"_n(x, y)+eta
  // If T'n(x, y) < Tc and Ttn(x, y) > Tc then Tt+1_n(x, y) = T"_n(x, y)-eta
  //
  // Gets samples from 3 texture stages:
  // tex0 = Center texel: T"(x,y) // i.e. set texcoord to (x,y)
  // tex1 = Result of LH pass 1.  // i.e. set texcoord to (x,y)
  // tex2 = Result of LH pass 2.  // i.e. set texcoord to (x,y)
  // tex3 = unused
  //
  // result is the sum of tex0 + 0.5*(tex1 + tex2), scaled and biased to
  // lie in [0, 1]
  nvparse("!!RC1.0                        \n"
          "{                              \n"
          "  rgb                          \n"
          "  {                            \n"
          "    discard = expand(tex1);    \n"
```

```
            "    discard = expand(tex2);    \n"
            "    spare0 = sum();            \n"
            "    scale_by_one_half();       \n"
            "    // average these.  range [-1,1] \n"
            "  }                            \n"
            "}                              \n"
            "{                              \n"
            "  rgb                          \n"
            "  {                            \n"
            "    discard = tex0;            \n"
            "    discard = spare0;          \n"
            "    spare0 = sum();            \n"
            "    // add.  range [-1,2] (but just clamp to [-1, 1]) \n"
            "  }                            \n"
            "}                              \n"
            "out.rgb = spare0;              \n"
            "out.a = unsigned_invert(zero); \n");

    // This texture shader script sets up the texture units to do normal
    // 2D texture operations to provide the three inputs needed by the above
    // register combiners script.
    nvparse("!!TS1.0\n"
            "texture_2d();\n"
            "texture_2d();\n"
            "texture_2d();\n"
            "nop();\n");

    glEnable(GL_TEXTURE_SHADER_NV);
    glEnable(GL_REGISTER_COMBINERS_NV);

    glActiveTextureARB(GL_TEXTURE0_ARB);
    glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_BUOYANCY_2]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glActiveTextureARB(GL_TEXTURE1_ARB);
    glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_LATENT_HEAT_1]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glActiveTextureARB(GL_TEXTURE2_ARB);
    glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_LATENT_HEAT_2]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

    RenderQuad(PASS_LATENT_HEAT_3);

    // Now we need to copy the resulting pixels into the output
    glBindTexture(GL_TEXTURE_2D, iOutputTextureIDs[PASS_LATENT_HEAT_3]);
    glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, iWidth, iHeight);
  }
  glEndList();
}
```