ภาคผนวก ข

โปรแกรมการปูลายผิวตามค่าพลศาสตร์ของไหล

การพัฒนาโปรแกรมสำหรับใช้ทดสอบการทำงานของการปูลายผิวตามค่าพลศาสตร์ของไหลนี้ ได้ประยุกต์ใช้โปรแกรมต้นแบบจากหนังสือ "*GPU Gems2 Programming Techniques for High-Performance Graphics and General-Purpose Computation*" ของสำนักพิมพ์ Addison-Wesley โดยเฉพาะในบทที่ 12 "*Tile-Based Texture Mapping*" ซึ่งโปรแกรมสามารถหาได้จาก *http://http.download.nvidia.com/developer/GPU_Gems_2/CD/Content/12.zip* มาเป็นต้นแบบโปรแกรม สำหรับใช้ในส่วนของการสังเคราะห์ และวิธีการปูลายผิวแวงไทล์ โดยทำการปรับปรุงโปรแกรม ให้สามารถแสดงผลลัพธ์ของลายผิวตามค่าพลศาสตร์ของไหล ด้วยการแก้ไขโปรแกรม TileMapping.cpp และ CgWangTileT.cpp

การปรับปรุงโปรแกรม TileMapping.cpp

```
/*
* TileMapping.cpp
*
* Li-Yi Wei
* 8/9/2003
*
* Modify by Mr.Supachai 2010 *
*/

#pragma warning (disable: 4786)

#ifdef WIN32
#include <windows.h>
#endif

#include <GL/gl.h>
#include <GL/glut.h>
#include <GL/glext.h>

#include <iostream>
#include <fstream>
#include <strstream>
#include <iomanip>
#include <vector>
#include <deque>

#include <math.h>
#include <ctime>
#include <time.h>
#include <assert.h>
using namespace std;

#include "WangTiles.hpp"
```

```
#include "WangTilesProcessor.hpp"
#include "CgWangTilesT.hpp"
#include "CgWangTilesC.hpp"

#include "Timer.hpp"
#include "EventTimer.hpp"
#include "FrameBuffer.hpp"

#include <glh/glh_extensions.h>

#define REQUIRED_EXTENSIONS "WGL_ARB_pbuffer " \
"WGL_ARB_pixel_format " \
"GL_NV_float_buffer " \
"GL_ARB_multitexture "

class ExitException : public Exception
{
    public:
    ExitException(const string & message);
};

ExitException::ExitException(const string & message) : Exception(message)
{
// nothing to do
}

CgWangTilesT * pCgWangTilesT = 0;
CgWangTilesC * pCgWangTilesC = 0;

class P4
{
    public:
    float x, y, z, w;
};

class TileCache
{
public:
TileCache(void) : _cacheLines(0) {};
~TileCache(void) {};

int Get(const WangTiles::Tile & tile, vector< vector<P4> > & image) const
{
    int index = Index(tile);

    if(index >= 0)
    {
            image = _cacheLines[index].image;
            return 1;
    }
    else
    {
            return 0;
    }
};

int Put(const WangTiles::Tile & tile, const vector< vector<P4> > & image)
{
    int index = Index(tile);

    if(index < 0)
    {
            Entry newGuy;
```

```
		newGuy.tile = tile;
		_cacheLines.push_back(newGuy);
		index = _cacheLines.size() - 1;
	}

	assert(index >= 0);

	_cacheLines[index].image = image;
	return 1;
};

int TileHeight(void) const
{
	if(_cacheLines.size() > 0)
	{
		return _cacheLines[0].image.size();
	}
	else
	{
		return 0;
	}
};

int TileWidth(void) const
{
	if(_cacheLines.size() > 0)
	{
		return _cacheLines[0].image[0].size();
	}
	else
	{
		return 0;
	}
};

protected:
int Index(const WangTiles::Tile & tile) const
{
	int result = -1;
	for(unsigned int i = 0;i < _cacheLines.size();i++)
	{
		if(tile.ID() == _cacheLines[i].tile.ID())
		{
			result = i;
		}
	}

	return result;
};

struct Entry
{
	WangTiles::Tile tile;
	vector< vector<P4> > image;
};

deque<Entry> _cacheLines;
};

struct CameraCoord
{
	CameraCoord(void)
	{
		from[0] = from[1] = from[2] = from[3] = 0;
		to[0] = to[1] = to[2] = to[3] = 0;
	}

	CameraCoord(const float from0,
	const float from1,
	const float from2,
	const float from3,
	const float to0,
	const float to1,
	const float to2,
	const float to3)
{
	from[0] = from0;to[0] = to0;
	from[1] = from1;to[1] = to1;
	from[2] = from2;to[2] = to2;
	from[3] = from3;to[3] = to3;
}

float from[4], to[4];
};

class GlobalParameters
{
	public:
	int resultWindow, tileWindow, cornerWindow, timerWindow;

	GLuint tilesTextureID;
	GLuint cornersTextureID;
	GLuint tileMappingTextureID;
	GLuint cornerMappingTextureID;
	GLuint resultTextureID;
	GLuint permutationTextureID;

	int numHColors;
	int numVColors;
	int numTilesPerColor;

	int tileSize;

	int mappingTextureHeight;
	int mappingTextureWidth;

	int permutationTextureSize;

	int reset;

	int compaction;// 0 for random, 1 for even

	float cornerSharpness;

	Timer timer;
	int eventHistory;
	EventTimer *pEventTimer;

	float startTime, endTime;
	int numFrames;

	WangTiles::TileSet *pTileSet;

	TileCache tileCache;
```

```
  int changeTileCache;

  vector< vector<WangTiles::Tile> > tileCompaction;
  vector< vector<WangTiles::Tile> > cornerCompaction;

  vector< vector<WangTiles::Tile> > tileMapping;
  vector< vector<WangTiles::Tile> > cornerMapping;

  // scene options
  int perspectiveView;// 1 or 0 control signal
  int winHeight, winWidth;
  float fov, eye[3], center[3];

  // visualization path
  deque<CameraCoord> cameraPath;
  int cameraPathIndex;
  int dumpAnimation;

  int sanitize;
};

GlobalParameters global;

void AddLinearPath(const int numFrames,
const CameraCoord & start,
const CameraCoord & end,
deque<CameraCoord> & path)
{
  CameraCoord delta;
{
for(int i = 0;i < 4;i++)
{
  delta.from[i] = (end.from[i] - start.from[i])/(numFrames-1);
  delta.to[i] = (end.to[i] - start.to[i])/(numFrames-1);
}
}

CameraCoord current = start;

for(int i = 0;i < numFrames;i++)
{
  path.push_back(current);

  for(int j = 0;j < 4;j++)
  {
          current.from[j] += delta.from[j];
          current.to[j] += delta.to[j];
  }
}
}

void BuildVisualizationPath(void)
{
// parameters
  const int numZoomFrames = 100;
  const int numPitchFrames = 50;
  const int numMoveFrames = 200;

  const float zFar = 2.0;
  const float zNear = 0.03;
  const float yShift = -0.15;
  const float yPitch = 0.06;
```

```
   const float yPitch1 = 0.32;
   const CameraCoord zoomOut(0, yShift, zFar, 0, 0, yShift, 0, 0);
   const CameraCoord zoomIn(0, yShift, zNear, 0, 0, yShift, 0, 0);
   const CameraCoord p1(0, yShift, zNear, 0, 0, yPitch+yShift, 0, 0);
   const CameraCoord p2(0, yPitch1+yShift, zNear, 0, 0, yPitch+yPitch1+yShift, 0, 0);

// reset
   global.cameraPath.clear();
   global.cameraPathIndex = 0;

// zoom in
   AddLinearPath(numZoomFrames, zoomOut, zoomIn, global.cameraPath);
// pitch up
   AddLinearPath(numPitchFrames, zoomIn, p1, global.cameraPath);
// forward
   AddLinearPath(numMoveFrames, p1, p2, global.cameraPath);

   if(!global.dumpAnimation)
   {
   // backward
         AddLinearPath(numMoveFrames, p2, p1, global.cameraPath);
   // pitch down
         AddLinearPath(numPitchFrames, p1, zoomIn, global.cameraPath);
   // zoom out
         AddLinearPath(numZoomFrames, zoomIn, zoomOut, global.cameraPath);
   }
}

void RenderQuad(const float centerX, const float centerY)
{
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glBegin(GL_QUADS);

   glColor3f(0.0, 0.0, 1.0);// blue
   glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0, 0);
   glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0, 0);
   glVertex2f(centerX - 1, centerY - 1);

   glColor3f(0.0, 1.0, 0.0);// green
   glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1, 0);
   glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1, 0);
   glVertex2f(centerX + 1, centerY - 1);

   glColor3f(1.0, 0.0, 0.0);// red
   glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1, 1);
   glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1, 1);
   glVertex2f(centerX + 1, centerY + 1);

   glColor3f(1.0, 1.0, 0.0);// yellow
   glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0, 1);
   glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0, 1);
   glVertex2f(centerX - 1, centerY + 1);

   glEnd();
}

void Enable(void)
```

```
{
  if(pCgWangTilesT) pCgWangTilesT->Enable();
  if(pCgWangTilesC) pCgWangTilesC->Enable();
}

void Disable(void)
{
  if(pCgWangTilesT) pCgWangTilesT->Disable();
  if(pCgWangTilesC) pCgWangTilesC->Disable();
}

void Display(void)
{
if( (glutGetWindow() == global.resultWindow) &&
  (global.resultTextureID <= 0) )
{
  if(global.tileMappingTextureID > 0)
  {
    glActiveTextureARB(GL_TEXTURE0_ARB);
    glBindTexture(GL_TEXTURE_RECTANGLE_NV, global.tileMappingTextureID);
  glEnable(GL_TEXTURE_RECTANGLE_NV);
  }

  if(global.cornerMappingTextureID > 0)
  {
  glActiveTextureARB(GL_TEXTURE1_ARB);
  glBindTexture(GL_TEXTURE_RECTANGLE_NV, global.cornerMappingTextureID);
  glEnable(GL_TEXTURE_RECTANGLE_NV);
  }

  Enable();
}
else // tileWindow or cornerWindow or timerWindow
{
  Disable();
}

if(global.tilesTextureID > 0)
{
  glActiveTextureARB(GL_TEXTURE0_ARB);
  glBindTexture(GL_TEXTURE_2D, global.tilesTextureID);
  if(glutGetWindow() != global.cornerWindow)
  {
         glEnable(GL_TEXTURE_2D);
  }
}

if(global.cornersTextureID > 0)
{
  glActiveTextureARB(GL_TEXTURE1_ARB);
  glBindTexture(GL_TEXTURE_2D, global.cornersTextureID);
  if(glutGetWindow() != global.tileWindow)
  {
         glEnable(GL_TEXTURE_2D);
  }
}

if(global.resultTextureID > 0)
{
  if(glutGetWindow() == global.resultWindow)
  {
  glActiveTextureARB(GL_TEXTURE1_ARB);
  glBindTexture(GL_TEXTURE_2D, global.cornersTextureID);
```

```
   glDisable(GL_TEXTURE_2D);

   glActiveTextureARB(GL_TEXTURE0_ARB);
   glBindTexture(GL_TEXTURE_2D, global.resultTextureID);
   glEnable(GL_TEXTURE_2D);
   }
}

if((global.cameraPath.size()) > 0 && (glutGetWindow() == global.resultWindow))
{
   // projection
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(global.fov,
   1.0*global.winWidth/global.winHeight,
   0.01, 100.0);

   // lookat
   glMatrixMode(GL_MODELVIEW);

   glPopMatrix();
   glPushMatrix();

   const CameraCoord & camera = global.cameraPath[global.cameraPathIndex];
   global.cameraPathIndex = (global.cameraPathIndex+1)%global.cameraPath.size();

   if(global.dumpAnimation && (global.cameraPathIndex == 0))
   {
          throw ExitException("done animation dumping");
   }

   gluLookAt(camera.from[0], camera.from[1], camera.from[2],
   camera.to[0],
   camera.to[1],
   camera.to[2],
   0, 1, 0);

   glTranslatef(0.0, 0.0, -camera.from[2]);
}

if(global.perspectiveView && (glutGetWindow() == global.resultWindow))
{
   // projection
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(global.fov,
   1.0*global.winWidth/global.winHeight,
   0.1, 10.0);

   // zoom
   glMatrixMode(GL_MODELVIEW);

   glPopMatrix();
   glPushMatrix();

   gluLookAt(global.eye[0], global.eye[1], global.eye[2],
   global.center[0],
   global.center[1],
   global.center[2],
  0, 1, 0);

  glTranslatef(0.0, 0.0, -global.eye[2]);
}

RenderQuad(0, 0);

if(global.pEventTimer && (glutGetWindow() == global.resultWindow))
{
  global.pEventTimer->RecordTime(global.timer.CurrentTime());
}

if(global.startTime <= 0)
{
  global.startTime = global.timer.CurrentTime();
}

if(glutGetWindow() == global.resultWindow)
{
  global.numFrames++;
}

if(global.dumpAnimation)
{
  glReadBuffer(GL_BACK);
  glFlush();

  int params[6];
  glGetIntegerv(GL_VIEWPORT, params);
  glutSetCursor(GLUT_CURSOR_WAIT);
  char filename[256];
  sprintf(filename, "frame_%.4d.ppm", global.cameraPathIndex);
  FrameBuffer::WriteColor(params[2], params[3], filename);
  glutSetCursor(GLUT_CURSOR_INHERIT);
}

glutSwapBuffers();
}

void DisplayString(GLfloat x, GLfloat y, GLfloat scale, const string &
message)
{
  glPushMatrix();

  glTranslatef(x, y, 0);
  glScalef(scale, scale, scale);

  for(int i = 0;i < message.length();i++)
  glutStrokeCharacter(GLUT_STROKE_ROMAN, message[i]);

  glPopMatrix();
}

void TimerDisplay(void)
{
  if(global.pEventTimer)
  {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  strstream strResult;

  float frameRate = 0;
```

```
    float elapsedTime = global.pEventTimer->ElapsedTime(-1,
global.eventHistory);

    if(elapsedTime > 0)
    {
            frameRate = global.eventHistory/elapsedTime;
    }

    strResult << frameRate;

    string message(strResult.str(), strResult.pcount());
    strResult.rdbuf()->freeze(0);

    glColor3f(1.0, 1.0, 1.0);
    DisplayString(-0.75, 0, 0.002, message);

    glutSwapBuffers();
    }
}

// colors contain 5 components:
// colors for bottom, right, top, and left edges,
// and central color
int CreateDiamondTile(const vector<P4> & colors,
const int height, const int width,
vector< vector<P4> > & result)
{
    if(colors.size() < 5)
    {
            return 0;
    }

    if(result.size() != height)
    {
            result = vector< vector<P4> >(height);
    }

{
for(int i = 0;i < result.size();i++)
{
    if(result[i].size() != width)
    {
            result[i] = vector<P4>(width);
    }
}
}

{
for(int i = 0;i < height;i++)
for(int j = 0;j < width;j++)
{
    int eq1 = height * j - width * i;
    int eq2 = height * j + width * i - height*width;

    P4 color;

    if( (i > height/16) && (i < 15*height/16) &&
    (j > width/16) && (j < 15*width/16) )
            color = colors[4];
    else if((eq1 > 0) && (eq2 <= 0)) color = colors[0];
    else if((eq1 > 0) && (eq2 > 0)) color = colors[1];
    else if((eq1 <= 0) && (eq2 > 0)) color = colors[2];
```

```
   else color = colors[3];

   result[i][j] = color;
}
}

// done
return 1;
}

void Randomize(vector<P4> & colors)
{
   float tmp[5][4] = { {1.0,0.2,0.2,0.0},
   {0.2,1.0,0.2,0.0},
   {0.2,0.2,1.0,0.0},
   {0.8,0.8,0.2,0.0},
   {0.2,0.8,0.8,0.0}};
   for(int i = 0;i < colors.size();i++)
   {
         colors[i].x = tmp[i%5][0];
         colors[i].y = tmp[i%5][1];
         colors[i].z = tmp[i%5][2];
         colors[i].w = tmp[i%5][3];
   }
}

int TileCompaction(vector< vector<WangTiles::Tile> > & result,
const int compaction_method)
{
   int status;

   if(! global.pTileSet)
   {
         return 0;
   }

   if(compaction_method == 1)
   {
   status = WangTiles::OrthogonalCompaction(*global.pTileSet, result);
   }
   else
   {
         status = WangTiles::RandomCompaction(*global.pTileSet, result);
   }

   return status;
}

int CornerCompaction(vector< vector<WangTiles::Tile> > & result)
{
   int status;

   status = WangTiles::OrthogonalCornerCompaction(*global.pTileSet,
result);

   return status;
}

int CreateDiamondTileSet(const int numHColors,
   const int numVColors,
   const int numTilesPerColor,
   const int tileSize,
   TileCache & cache)
```

```
{
    if(global.tileCompaction.size() <= 0)
    {
    return 0;
    }

    const int tileTextureHeight = global.tileCompaction.size();
    const int tileTextureWidth = global.tileCompaction[0].size();

    // random tile colors
    vector<P4> hColors(numHColors);
    vector<P4> vColors(numVColors);
    vector<P4> cColors(tileTextureHeight*tileTextureWidth);

    Randomize(hColors);
    Randomize(vColors);

    for(int i=0;i<cColors.size();i++){
    float temp = rand()*1.0/RAND_MAX;
    cColors[i].w = temp;
    cColors[i].x = 0.9;
    cColors[i].y = 0.9;
    cColors[i].z = 0.9;
}

{
for(int i = 0;i < tileTextureHeight;i++)
for(int j = 0;j < tileTextureWidth;j++)
{
    // build the tile
    vector<P4> colors(5);

    WangTiles::Tile compactionTile = global.tileCompaction[i][j];

    colors[0] = hColors[compactionTile.e0()];
    colors[1] = vColors[compactionTile.e1()];
    colors[2] = hColors[compactionTile.e2()];
    colors[3] = vColors[compactionTile.e3()];
    colors[4] = cColors[compactionTile.ID()];

    vector< vector<P4> > tile;

    if(! CreateDiamondTile(colors, tileSize, tileSize, tile))
    {
            throw Exception("Error in CreateDiamondTile");
    }

    cache.Put(compactionTile, tile);
}
}

return 1;
}

int CreateInputTileSet(const vector< vector<FrameBuffer::P3> > &
inputPacking,
    const int maximumValue,
    const int numHColors,
    const int numVColors,
    const int numTilesPerColor,
    TileCache & cache)
{
    if(global.tileCompaction.size() <= 0)
    {
        return 0;
    }

    const int packingTextureHeight = inputPacking.size();
    const int packingTextureWidth = packingTextureHeight > 0 ? inputPacking[0].size(): 0;
    const int tileTextureHeight = global.tileCompaction.size();
    const int tileTextureWidth = global.tileCompaction[0].size();
    const int tileHeight = packingTextureHeight/tileTextureHeight;
    const int tileWidth = packingTextureWidth/tileTextureWidth;

    assert(packingTextureHeight%tileTextureHeight == 0);
    assert(packingTextureWidth%tileTextureWidth == 0);

{
    vector< vector<P4> > tile(tileHeight);
    {
        for(int i = 0; i < tileHeight; i++)
        {
            tile[i] = vector<P4>(tileWidth);
        }
    }

    vector< vector<WangTiles::Tile> > goodTileCompaction;

    if(! TileCompaction(goodTileCompaction, 1))
    {
    return 0;
    }

    for(int i = 0; i < tileTextureHeight; i++)
    for(int j = 0; j < tileTextureWidth; j++)
    {
    // build the tile from good compaction
    WangTiles::Tile compactionTile = goodTileCompaction[i][j];

    // assign to the tile texture
    for(int m = i*tileHeight; m < (i+1)*tileHeight; m++)
    for(int n = j*tileWidth; n < (j+1)*tileWidth; n++)
    {
        FrameBuffer::P3 inputColor = inputPacking[m][n];

        P4 outputColor;
        outputColor.x = inputColor.r*1.0/maximumValue;
        outputColor.y = inputColor.g*1.0/maximumValue;
        outputColor.z = inputColor.b*1.0/maximumValue;
        outputColor.w = 1.0;

        tile[m - i*tileHeight][n - j*tileWidth] = outputColor;
    }

    cache.Put(compactionTile, tile);
}
}

return 1;
}

struct ImageSize
{
  int height, width;
};

// return number of mipmap levels if successful, 0 else
// assuming RGBA pixel format, and data is of sufficient size
int SanitizeTileTexture(const vector< vector<WangTiles::Tile> > & tiles,
const int height, const int width,
float * data)
{
// error checking
  if((height <= 0) || (width <= 0))
  {
        return 0;
  }

  // initialization
  Array2D<int> pyramidSpec;
  {
  vector<ImageSize> sizeSpec;
  ImageSize current;
  current.height = height;current.width = width;

  do
  {
        sizeSpec.push_back(current);

        current.height /= 2;current.width /= 2;
  }
  while((current.height > 1) || (current.width > 1));

  pyramidSpec = Array2D<int>(sizeSpec.size(), 3);
  for(unsigned int i = 0;i < sizeSpec.size();i++)
  {
        pyramidSpec[i][0] = sizeSpec[i].height;
        pyramidSpec[i][1] = sizeSpec[i].width;
        // assuming ARGB pixel format
        pyramidSpec[i][2] = 4;
  }
}

ImagePyramid<float> pyramid(pyramidSpec);

{
  Array3D<float> & image = pyramid[0];

  int index = 0;
  for(int row = 0;row < image.Size(0);row++)
  for(int col = 0;col < image.Size(1);col++)
  for(int cha = 0;cha < image.Size(2);cha++)
  {
        image[row][col][cha] = data[index++];
  }
}

// sanitize
if(! WangTilesProcessor::BoxSanitize(tiles, pyramid, pyramid,
global.sanitize ==
2))
{
  return 0;
}

```

```
{
    const int mismatch_level = WangTilesProcessor::TileMismatch(tiles, pyramid);
    if(mismatch_level >= 0)
    {
            cerr << "tile mismatch at level " << mismatch_level << endl;
    }
}

// copy to output
{
int index = 0;

for(int level = 0;level < pyramid.NumLevels();level++)
{
    Array3D<float> & image = pyramid[level];

    for(int row = 0;row < image.Size(0);row++)
    for(int col = 0;col < image.Size(1);col++)
    for(int cha = 0;cha < image.Size(2);cha++)
    {
            data[index++] = image[row][col][cha];
    }
}
}

// done
return pyramid.NumLevels();
}

int CreateTileTexture(const GLuint tileTextureID,
const GLuint cornerTextureID,
const int numHColors,
const int numVColors,
const int numTilesPerColor,
const int tileSize)
{
    // create the diamond tile set
    if(global.changeTileCache &&
    !CreateDiamondTileSet(numHColors, numVColors,
    numTilesPerColor, tileSize,
    global.tileCache))
    {
            throw Exception("error in CreateDiamondTileSet()");
    }

    if( (global.tileCompaction.size() <= 0) ||
    (global.cornerCompaction.size() <= 0) )
    {
            return 0;
    }

    const int tileTextureHeight = global.tileCompaction.size();
    const int tileTextureWidth = global.tileCompaction[0].size();

    const int cornerTextureHeight = global.cornerCompaction.size();
    const int cornerTextureWidth = global.cornerCompaction[0].size();

    // Fill in the tile texture map.
    const int glTileTextureHeight = tileSize *tileTextureHeight;
    const int glTileTextureWidth = tileSize * tileTextureWidth;
    float *data1 = new float[glTileTextureHeight*glTileTextureWidth*4*2];

```

```
  {
  for(int i = 0;i < tileTextureHeight;i++)
  for(int j = 0;j < tileTextureWidth;j++)
  {
  WangTiles::Tile compactionTile = global.tileCompaction[i][j];

  vector< vector<P4> > tile;

  if(! global.tileCache.Get(compactionTile, tile))
  {
        throw Exception("Error in tileCache.Get()");
  }

  // assign to the tile texture
  for(int m = i*tileSize;m < (i+1)*tileSize;m++)
  for(int n = j*tileSize;n < (j+1)*tileSize;n++)
  {
        int op = m*glTileTextureWidth + n;
        P4 color = tile[m - i*tileSize][n - j*tileSize];

        data1[4*op + 0] = color.x;
        data1[4*op + 1] = color.y;
        data1[4*op + 2] = color.z;
        data1[4*op + 3] = color.w;
  }
}
}

// Fill in the corner texture map.
const int glCornerTextureHeight = tileSize *cornerTextureHeight;
const int glCornerTextureWidth = tileSize * cornerTextureWidth;
float *data2 = new float[glCornerTextureHeight*glCornerTextureWidth*4*2];

if(cornerTextureID > 0)
{
  for(int i = 0;i < cornerTextureHeight;i++)
  for(int j = 0;j < cornerTextureWidth;j++)
  {
  // build the tile
  vector<P4> colors(5);

  WangTiles::Tile compactionTile = global.cornerCompaction[i][j];

  vector< vector<P4> > tile;

  if(! global.tileCache.Get(compactionTile, tile))
  {
        throw Exception("Error in tileCache.Get()");
  }

  // assign to the tile texture
  for(int m = i*tileSize;m < (i+1)*tileSize;m++)
  for(int n = j*tileSize;n < (j+1)*tileSize;n++)
  {
        int op = m*glCornerTextureWidth + n;
        P4 color = tile[m - i*tileSize][n - j*tileSize];

        data2[4*op + 0] = color.x;
        data2[4*op + 1] = color.y;
        data2[4*op + 2] = color.z;
        data2[4*op + 3] = color.w;
  }
```

```
   }
}

// valid window IDs
vector<int> validWins;

if(global.resultWindow) validWins.push_back(global.resultWindow);
if(global.tileWindow) validWins.push_back(global.tileWindow);
if(global.cornerWindow) validWins.push_back(global.cornerWindow);

int originalWin = glutGetWindow();

for(int i = 0;i < validWins.size();i++)
{
   glutSetWindow(validWins[i]);

   // tile texture
   glActiveTextureARB(GL_TEXTURE0_ARB);
   glBindTexture(GL_TEXTURE_2D, tileTextureID);
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

   glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, glTileTextureWidth, glTileTextureHeight, 0, GL_RGBA, GL_FLOAT, data1);

   if(global.sanitize)
   {
   const int num_levels = SanitizeTileTexture(global.tileCompaction, glTileTextureHeight, glTileTextureWidth, data1);

   if(num_levels <= 0)
   {
          throw Exception("error in SanitizeTileTexture!");
   return 0;
   }

   int offset = 0;
   int glTileTextureMipWidth = glTileTextureWidth;
   int glTileTextureMipHeight = glTileTextureHeight;

   for(int i = 0;i < num_levels;i++, offset += glTileTextureMipWidth* glTileTextureMipHeight*4, glTileTextureMipWidth/= 2, glTileTextureMipHeight/= 2)
   {
   glTexImage2D(GL_TEXTURE_2D, i, GL_RGBA, glTileTextureMipWidth, glTileTextureMipHeight, 0, GL_RGBA, GL_FLOAT, &data1[offset]);
   }
   }

   // corner texture
   if(cornerTextureID > 0)
   {
   glActiveTextureARB(GL_TEXTURE1_ARB);
   glBindTexture(GL_TEXTURE_2D, cornerTextureID);
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

```

```
  glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, glCornerTextureWidth, glCornerTextureHeight, 0, GL_RGBA, GL_FLOAT, data2);

  if(global.sanitize)
  {
  const int num_levels = SanitizeTileTexture(global.cornerCompaction, glTileTextureHeight, glTileTextureWidth, data2);

  if(num_levels <= 0)
  {
        throw Exception("error in Sanitize CornerTexture!");
        return 0;
  }

  int offset = 0;
  int glCornerTextureMipWidth = glCornerTextureWidth;
  int glCornerTextureMipHeight = glCornerTextureHeight;

  for(int i = 0;i < num_levels;i++, offset += glCornerTextureMipWidth *glCornerTextureMipHeight*4, glCornerTextureMipWidth/= 2, glCornerTextureMipHeight/= 2)
  {
  glTexImage2D(GL_TEXTURE_2D, i, GL_RGBA, glCornerTextureMipWidth, glCornerTextureMipHeight, 0, GL_RGBA, GL_FLOAT, &data2[offset]);
  }
  }
}
}

glutSetWindow(originalWin);

delete[] data1;
delete[] data2;

return 1;
}

int CreateTileTexture(const int numHColors,
  const int numVColors,
  const int numTilesPerColor,
  const int tileSize,
  GLuint & tileTextureID,
  GLuint & cornerTextureID
  )
{
  glGenTextures(1, &tileTextureID);
  if(global.cornerSharpness > 0)
  {
        glGenTextures(1, & cornerTextureID);
  }

  return CreateTileTexture(tileTextureID, cornerTextureID,
  numHColors, numVColors, numTilesPerColor,
  tileSize);
}

```

```
void DumpMapping(const vector< vector<WangTiles::Tile> > & mapping)
{
  const int height = mapping.size();
  const int width = mapping[0].size();

  for(int i = height - 1;i >= 0;i--)
  {
  {
          for(int j = 0;j < width;j++)
          {
          cout << " " << mapping[i][j].e2() << " ";
          }
  }

  cout << endl;

  {
  for(int j = 0;j < width;j++)
          {
          cout << mapping[i][j].e3() << " " << mapping[i][j].e1() << " ";
          }
  }

  cout << endl;

  {
  for(int j = 0;j < width;j++)
  {
          cout << " " << mapping[i][j].e0() << " ";
  }
  }

  cout << endl;
}
}

int CreateMappingTexture(const GLuint tileMappingTextureID,
          const GLuint cornerMappingTextureID,
          const GLuint resultTextureID,
          const int numHColors,
          const int numVColors,
          const int numTilesPerColor,
          const int mappingTextureHeight,
          const int mappingTextureWidth,
          const int tileHeight,
          const int tileWidth,
          const int remapping)
{
// sequential tiling
  if(remapping ||
          (global.tileMapping.size() != mappingTextureHeight) ||
          (global.tileMapping[0].size() != mappingTextureWidth))
  {
          if(! WangTiles::SequentialTiling(*global.pTileSet,
                          mappingTextureHeight,
                          mappingTextureWidth,
                          global.tileMapping) )
          {
                  throw Exception("error in sequential tiling");
          }
  }

  if(global.tileCompaction.size() <= 0)
```

```
{
        return 0;
}

const int tileTextureHeight = global.tileCompaction.size();
const int tileTextureWidth = global.tileCompaction[0].size();

float *data = new float[mappingTextureHeight*mappingTextureWidth*4];

#if 0
        cout << "tile map" << endl;
        DumpMapping(global.tileMapping);

#endif
// loading LBM's Kinetics Energy.
double *lbm = new double[mappingTextureHeight*mappingTextureWidth];
ifstream lbm_file("KineticEnergy.txt");

if(!lbm_file){
cout << endl << "Open LBM file Failure, It will be random value." << endl;
for(int i = 0;i< mappingTextureHeight * mappingTextureWidth;i++)
lbm[i] = rand()*1.0/RAND_MAX;
}else{
double lbm_max, lbm_min;
lbm_file >> lbm_max >> lbm_min;
for(int j = 0;j< mappingTextureWidth;j++)
for(int i = 0;i < mappingTextureHeight;i++)
{
        int k = i*mappingTextureWidth + j;
        lbm_file >> lbm[k];
        lbm[k] = (lbm_max - lbm[k]) / (lbm_max - lbm_min);
}

lbm_file.close();
}


// tile mapping texture
{
glActiveTextureARB(GL_TEXTURE0_ARB);
glBindTexture(GL_TEXTURE_RECTANGLE_NV, tileMappingTextureID);
glTexParameteri(GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

for(int i = 0;i < mappingTextureHeight;i++)
for(int j = 0;j < mappingTextureWidth;j++)
{
int id = global.tileMapping[i][j].ID();
int row, col;

if(WangTiles::TileLocation(global.tileCompaction, id, row, col) != 1)
{
throw Exception("error in WangTiles::TileLocation()");
}

int op = i*mappingTextureWidth + j;

data[4*op + 0] = col;
```

```
  data[4*op + 1] = row;
  data[4*op + 2] = 0;
  data[4*op + 3] = float(lbm[op]);//0.9;
  }

  glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGBA_NV, mappingTextureWidth, mappingTextureHeight, 0, GL_RGBA, GL_FLOAT, data);
  }

// corner tiling texture
  if(cornerMappingTextureID > 0)
  {
  // create a corner compaction
  if(global.cornerCompaction.size() <= 0)
  {
  return 0;
  }

  // corner tiling
  vector< vector<WangTiles::Tile> > cornerMapping;

  if(! WangTiles::ShiftedCornerTiling(global.cornerCompaction,
  global.tileMapping,
  global.cornerMapping) )
  {
  throw Exception("error in shifted corner tiling");
  }
#if 0
  //cout << "tile map" << endl;
  //DumpMapping(global.tileMapping);

  cout << endl << "corner map" << endl;
  DumpMapping(global.cornerMapping);
#endif
  glActiveTextureARB(GL_TEXTURE1_ARB);
  glBindTexture(GL_TEXTURE_RECTANGLE_NV, cornerMappingTextureID);
  glTexParameteri(GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

  for(int i = 0;i < mappingTextureHeight;i++)
  for(int j = 0;j < mappingTextureWidth;j++)
  {
  int e0 = global.tileMapping[i][j].e1();
  int e1 = global.tileMapping[i][(j+1)%mappingTextureWidth].e2();
  int e2 = global.tileMapping[(i+1)%mappingTextureHeight][(j+1)%mappingTextureWidth].e3();
  int e3 = global.tileMapping[(i+1)%mappingTextureHeight][j].e0();

  int row, col;

  if(WangTiles::CornerLocation(global.cornerCompaction,
          e0, e1, e2, e3, row, col) <= 0)
  {
  throw Exception("error in WangTiles::CornerLocation()");
  }

  int op = i*mappingTextureWidth + j;

  data[4*op + 0] = col;
  data[4*op + 1] = row;
```

```
data[4*op + 2] = 0;
data[4*op + 3] = 0;
}

glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGBA_NV,
mappingTextureWidth, mappingTextureHeight, 1, GL_RGBA, GL_FLOAT, data);
}

delete[] data;
delete[] lbm;

if(resultTextureID > 0)
{
const int numVTiles = mappingTextureHeight;
const int numHTiles = mappingTextureWidth;

const int resultTextureHeight = numVTiles * tileHeight;
const int resultTextureWidth = numHTiles * tileWidth;

float *data = new float[resultTextureHeight*resultTextureWidth*4];

glActiveTextureARB(GL_TEXTURE0_ARB);
glBindTexture(GL_TEXTURE_2D, resultTextureID);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

for(int i = 0; i < numVTiles; i++)
for(int j = 0; j < numHTiles; j++)
{
// build the tile
vector<P4> colors(5);

int id = global.tileMapping[i][j].ID();
int row, col;

if(WangTiles::TileLocation(global.tileCompaction, id, row, col) != 1)
{
throw Exception("error in WangTiles::TileLocation()");
}

WangTiles::Tile compactionTile = global.tileCompaction[row][col];

vector< vector<P4> > tile;

if(! global.tileCache.Get(compactionTile, tile))
{
throw Exception("Error in tileCache.Get()");
}

// assign to the result texture
for(int m = i*tileHeight; m < (i+1)*tileHeight; m++)
for(int n = j*tileWidth; n < (j+1)*tileWidth; n++)
{
int op = m*resultTextureWidth + n;
P4 color = tile[m - i*tileHeight][n - j*tileWidth];

```

```
  data[4*op + 0] = color.x;
  data[4*op + 1] = color.y;
  data[4*op + 2] = color.z;
  data[4*op + 3] = color.w;
  }
  }

  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, resultTextureWidth,
resultTextureHeight, 0, GL_RGBA, GL_FLOAT, data);

  delete[] data;
  }

  return 1;
}

int CreateMappingTexture(const int numHColors,
        const int numVColors,
        const int numTilesPerColor,
        const int mappingTextureHeight,
        const int mappingTextureWidth,
        const int tileHeight,
        const int tileWidth,
        GLuint & tileMappingTextureID,
        GLuint & cornerMappingTextureID,
        GLuint & resultTextureID)
{
  glGenTextures(1, &tileMappingTextureID);

  if(global.cornerSharpness > 0)
  {
        glGenTextures(1, &cornerMappingTextureID);
  }

  if(global.permutationTextureSize < 0)
  {
        glGenTextures(1, &resultTextureID);
  }

  return CreateMappingTexture(tileMappingTextureID,
        cornerMappingTextureID,
        resultTextureID,
        numHColors, numVColors, numTilesPerColor,
        mappingTextureHeight, mappingTextureWidth,
        tileHeight, tileWidth, 1);
}

int CreatePermutationTexture(const GLuint permutationTextureID,
const int permutationTextureSize)
{

  glBindTexture(GL_TEXTURE_RECTANGLE_NV, permutationTextureID);
  glTexParameteri(GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_WRAP_T, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
  glTexParameteri(GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);

  float *permutationData = new float[permutationTextureSize*4];

  vector<float> sequence(permutationTextureSize);
  {
```

```
  for(int i = 0;i < sequence.size();i++)
  {
        sequence[i] = i;
  }
  }

  // random permutation
  for(int i = 0;i < permutationTextureSize;i++)
  {
  int selection = rand()%sequence.size();

  permutationData[4*i] = permutationData[4*i + 1] = permutationData[4*i
+ 2] =
permutationData[4*i + 3] = sequence[selection];

  sequence[selection] = sequence[sequence.size() - 1];
  sequence.pop_back();
  }

  glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGBA_NV,
permutationTextureSize, 0, 0, GL_RGBA, GL_FLOAT, permutationData);

  delete[] permutationData;

  return permutationTextureID;
}

int CreatePermutationTexture(const int permutationTextureSize)
{
  GLuint handle;
  glGenTextures(1, &handle);

  return CreatePermutationTexture(handle, permutationTextureSize);
}

void Keyboard(unsigned char key, int x, int y)
{
  switch(key)
  {
  case 'D':
  {
        int params[6];
        glGetIntegerv(GL_VIEWPORT, params);
        glutSetCursor(GLUT_CURSOR_WAIT);
        FrameBuffer::WriteColor(params[2], params[3], "snapshot.ppm");
        glutSetCursor(GLUT_CURSOR_INHERIT);
  }
  break;

  case 27:
        throw ExitException("exit");
  break;

  case 'r':
  case 'R':
  if(! TileCompaction(global.tileCompaction, key == 'r'))
  {
        // force random compaction
        throw Exception("wrong TileCompaction in Keyboard()");
  }

  // follow through

  default:
        CreateTileTexture(global.tilesTextureID,
              global.cornersTextureID,
              global.numHColors,
              global.numVColors,
              global.numTilesPerColor,
              global.tileSize);

  if(glutGetWindow() == global.resultWindow)
  {
        if(pCgWangTilesT)
        {
              if(! CreateMappingTexture(global.tileMappingTextureID,
                          global.cornerMappingTextureID,
                          global.resultTextureID,
                          global.numHColors,
                          global.numVColors,
                          global.numTilesPerColor,
                          global.mappingTextureHeight,
                          global.mappingTextureWidth,
                          global.tileSize,
                          global.tileSize,
                          (key != 'r') && (key != 'R')))
        {
              throw Exception("error in CreateMappingTexture");
        }
        }

        if(pCgWangTilesC)
        {
        if(! CreatePermutationTexture(global.permutationTextureID,
              global.permutationTextureSize))
        {
        throw Exception("error in CreatePermutationTexture");
        }
        }
        }

        if(global.resultWindow > 0)
        glutPostWindowRedisplay(global.resultWindow);
        if(global.tileWindow > 0)
        glutPostWindowRedisplay(global.tileWindow);
        if(global.cornerWindow > 0)
        glutPostWindowRedisplay(global.cornerWindow);

  break;
  }
}

void Idle()
{
  //if(glutGetWindow() == global.resultWindow)
  {
  if(global.timerWindow > 0)
        glutPostWindowRedisplay(global.timerWindow);

  if(global.resultWindow > 0)
        glutPostWindowRedisplay(global.resultWindow);
  }
}

int Main(int argc, char **argv)
{
```

```
int argCtr = 0;
const int resultWinHeight = -1;
const int resultWinWidth = -1;
const int numVColors = 2;
const int numHColors = 2;
const int numTilesPerColor = 1;
const char * tilePackingFileName = "cloud128.ppm";
//const char * tilePackingFileName = "128";
int tileSize = atoi(tilePackingFileName);
const int mappingTextureHeight = 16;
const int mappingTextureWidth = 16;
const float cornerSharpness = 0.0;
const int hashTableSize = 0;
const int eventTableSize = -1;
const int eventHistorySize = 0;
const int compactionMethod = 1;
const int viewOption = 0;
const int sanitize = 0;
const int randomSeed = -1;

srand(randomSeed >= 0 ? randomSeed : time(0));

{
global.dumpAnimation = (viewOption == 3);

global.tilesTextureID = 0;
global.tileMappingTextureID = 0;
global.cornerMappingTextureID = 0;
global.resultTextureID = 0;

global.numHColors = numHColors;
global.numVColors = numVColors;
global.numTilesPerColor = numTilesPerColor;

global.tileSize = tileSize;
global.mappingTextureHeight = mappingTextureHeight;
global.mappingTextureWidth = mappingTextureWidth;
global.permutationTextureSize = hashTableSize;

global.reset = 0;

global.compaction = compactionMethod;

global.cornerSharpness = cornerSharpness;

global.eventHistory = eventHistorySize;

if(eventTableSize > 0)
{
      global.pEventTimer = new EventTimer(eventTableSize);
}
else
{
      global.pEventTimer = 0;
}

if((viewOption == 2) || (viewOption == 3)) BuildVisualizationPath();

global.startTime = global.endTime = -1;
global.numFrames = 0;

global.pTileSet = new WangTiles::TileSet(numHColors,
        numVColors,
        numTilesPerColor);

if(!TileCompaction(global.tileCompaction, global.compaction))
{
    throw Exception("error in creating compaction");
}

if(!CornerCompaction(global.cornerCompaction))
{
throw Exception("error in creating corner compaction");
}

global.resultWindow = global.tileWindow = global.cornerWindow = global.
timerWindow = 0;

if(tileSize <= 0)
{
    int maximumValue = 0;

    vector< vector<FrameBuffer::P3> > inputTilePacking;

    if(!FrameBuffer::ReadPPM(tilePackingFileName,
            inputTilePacking, maximumValue))
    {
        throw Exception("error in reading tile packing");
    }

    if(!CreateInputTileSet(inputTilePacking,
            maximumValue,
            numHColors,
            numVColors,
            numTilesPerColor,
            global.tileCache))
    {
        throw Exception("error in creating input tile set");
    }

if(global.tileCache.TileHeight() == global.tileCache.TileWidth())
{
    tileSize = global.tileCache.TileHeight();
}
else
{
    throw Exception("tileHeight != tileWidth");
}

global.tileSize = tileSize;
global.changeTileCache = 0;
}
else
{
global.changeTileCache = 1;
}

// scene
global.perspectiveView = (viewOption == 1);

global.winWidth = resultWinWidth > 0? resultWinWidth : tileSize *
mappingTextureWidth;
global.winHeight = resultWinHeight > 0? resultWinHeight : tileSize *
```

```
mappingTextureHeight;

global.fov = 45;

global.eye[0] = 0;global.eye[1] = -1;global.eye[2] = 0.1;

global.center[0] = 0;global.center[1] = -0.7;global.center[2] = 0;

global.sanitize = sanitize;
}

glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);

// tile-pack window
glutInitWindowSize(tileSize * global.tileCompaction[0].size(),
tileSize * global.tileCompaction.size());
global.tileWindow = glutCreateWindow("Wang Tiles");

glutKeyboardFunc(Keyboard);
glutDisplayFunc(Display);

// corner-pack window
if(global.cornerSharpness > 0)
{
glutInitWindowSize(tileSize * global.cornerCompaction[0].size(),
tileSize * global.cornerCompaction.size());
global.cornerWindow = glutCreateWindow("Corner Tiles");

glutKeyboardFunc(Keyboard);
glutDisplayFunc(Display);
}

// timer window
if(global.pEventTimer)
{
glutInitWindowSize(128, 128);
global.timerWindow = glutCreateWindow("Frame Rate");

glutKeyboardFunc(Keyboard);
glutDisplayFunc(TimerDisplay);
}

// result tiling window
glutInitWindowSize(global.winWidth, global.winHeight);
global.resultWindow = glutCreateWindow("Tiling");

glutKeyboardFunc(Keyboard);
glutDisplayFunc(Display);
if(global.timerWindow || (global.cameraPath.size() > 0))
{
glutIdleFunc(Idle);
}

if(!glh_init_extensions(REQUIRED_EXTENSIONS))
{
throw Exception("Necessary extensions were not supported");
}

glDisable(GL_LIGHTING);
glEnable(GL_DEPTH_TEST);

```

```
clock_t t1 = clock();
if(! CreateTileTexture(numHColors, numVColors, numTilesPerColor,
global.tileSize,
global.tilesTextureID, global.cornersTextureID))
{
throw Exception("error in creating tile textures");
}
clock_t t2 = clock();
cout << "Time CreateTileText: " << double(t2-
t1)/(double)CLOCKS_PER_SEC << endl;

if(hashTableSize > 0)
{
global.permutationTextureID = \
CreatePermutationTexture(hashTableSize);

pCgWangTilesC = new CgWangTilesC(global.tilesTextureID,
        global.cornersTextureID,
        tileSize, tileSize,
        numHColors*numHColors,
        numVColors*numVColors,
        numVColors*numVColors,
        numHColors*numHColors,
        mappingTextureHeight,
        mappingTextureWidth,
        global.permutationTextureID,
        cornerSharpness);
}
else
{
clock_t t3=clock();
if(! CreateMappingTexture(numHColors, numVColors, numTilesPerColor,
        mappingTextureHeight,
        mappingTextureWidth,
        tileSize,
        tileSize,
        global.tileMappingTextureID,
        global.cornerMappingTextureID,
        global.resultTextureID))
{
throw Exception("error in creating mapping textures");
}

if(hashTableSize == 0)
{
pCgWangTilesT = new CgWangTilesT(global.tilesTextureID,
global.cornersTextureID,
tileSize, tileSize,
global.tileMappingTextureID,
global.cornerMappingTextureID,
cornerSharpness);
}
else
{
pCgWangTilesT = 0;
}
clock_t t4=clock();
cout << "Time CreateMappingTexture: " << double(t4-
t3)/(double)CLOCKS_PER_SEC << endl;
}

Enable();

```

```
  glutMainLoop();

  if(pCgWangTilesT) delete pCgWangTilesT;
  if(pCgWangTilesC) delete pCgWangTilesC;

  return 0;
}

int main(int argc, char **argv)
{
  try
  {
        return Main(argc, argv);
  }
  catch(ExitException e)
  {
        if(global.pEventTimer)
        {
        // currently, the frame rate only makes sense if you turn on event timer.
        global.endTime = global.timer.CurrentTime();

        cout << "frame rate is " << global.numFrames/(global.endTime - global.startTime) << endl;
        }

        if(pCgWangTilesT) delete pCgWangTilesT;
        if(pCgWangTilesC) delete pCgWangTilesC;

        return 0;
  }
  catch(Exception e)
  {
        cerr<<"Error : "<<e.Message()<<endl;
        return 1;
  }
}

```

การปรับปรุงโปรแกรม CgWangTileT.cpp

```
/*
* CgWangTilesT.cpp
*
* Li-Yi Wei
* 8/9/2003
*
* Modify by Mr. Supachai 2010 *
*/

#include "CgWangTilesT.hpp"

#include <iostream>
#include <strstream>

using namespace std;

#include <glh/glh_extensions.h>

#define REQUIRED_EXTENSIONS "WGL_ARB_pbuffer " \
          "WGL_ARB_pixel_format " \
          "WGL_ARB_render_texture " \
          "GL_NV_float_buffer " \
          "GL_NV_texture_rectangle " \
          "GL_ARB_multitexture "

CgWangTilesT::CgWangTilesT(const GLuint tilesTextureID,
          const GLuint cornersTextureID,
          const int tileHeight,
          const int tileWidth,
          const GLuint tileMappingTextureID,
          const GLuint cornerMappingTextureID,
          const float cornerSharpness) throw(Exception) : _context(0)
, _program(0), _tilesTexture(0), _cornersTexture(0), _tileMappingTexture(0),
_cornerMappingTexture(0)
{
    if( ((tilesTextureID <= 0) && (tileMappingTextureID > 0)) ||
    ((tilesTextureID > 0) && (tileMappingTextureID <= 0)) ||
    ((cornersTextureID <= 0) && (cornerMappingTextureID > 0)) ||
    ((cornersTextureID > 0) && (cornerMappingTextureID <= 0)) )
    {
    throw Exception("CgWangTilesT: illegal tile or corner selection!");
    }

    cgSetErrorCallback(CheckError);

    _context = cgCreateContext();

    if(! _context)
    {
    throw Exception("null context");
    }

    _fragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
    cgGLSetOptimalOptions(_fragmentProfile);

    // tile texture size
    int tileTextureHeight = 0;
    int tileTextureWidth = 0;

    if(tilesTextureID > 0)
    {
          glActiveTextureARB(GL_TEXTURE0_ARB);
          glBindTexture(GL_TEXTURE_2D, tilesTextureID);

          glGetTexLevelParameteriv(GL_TEXTURE_2D, 0,
          GL_TEXTURE_HEIGHT, &tileTextureHeight);
          glGetTexLevelParameteriv(GL_TEXTURE_2D, 0,
          GL_TEXTURE_WIDTH, &tileTextureWidth);

          if( (tileTextureHeight <= 0) || (tileTextureWidth <= 0) )
          {
          throw Exception("CgWangTilesT: illegal tile texture size");
          }

          tileTextureHeight /= tileHeight;
          tileTextureWidth /= tileWidth;
    }

    // corner texture size
    int cornerTextureHeight = 0;
    int cornerTextureWidth = 0;

    if(cornersTextureID > 0)
    {
        glActiveTextureARB(GL_TEXTURE1_ARB);
        glBindTexture(GL_TEXTURE_2D, cornersTextureID);

        glGetTexLevelParameteriv(GL_TEXTURE_2D, 0,
            GL_TEXTURE_HEIGHT, &cornerTextureHeight);
        glGetTexLevelParameteriv(GL_TEXTURE_2D, 0,
            GL_TEXTURE_WIDTH, &cornerTextureWidth);

    if( (cornerTextureHeight <= 0) || (cornerTextureWidth <= 0) )
    {
        throw Exception("CgWangTilesT: illegal corner texture size");
    }

    cornerTextureHeight /= tileHeight;
    cornerTextureWidth /= tileWidth;
    }

    // tile mapping texture size
    int tileMappingTextureHeight = 0;
    int tileMappingTextureWidth = 0;

    if(tileMappingTextureID > 0)
    {
        glActiveTextureARB(GL_TEXTURE0_ARB);
        glBindTexture(GL_TEXTURE_RECTANGLE_NV, tileMappingTextureID);

        glGetTexLevelParameteriv(GL_TEXTURE_RECTANGLE_NV, 0,
            GL_TEXTURE_HEIGHT, &tileMappingTextureHeight);
        glGetTexLevelParameteriv(GL_TEXTURE_RECTANGLE_NV, 0,
            GL_TEXTURE_WIDTH, &tileMappingTextureWidth);

        if( (tileMappingTextureHeight <= 0) || (tileMappingTextureWidth <= 0) )
        {
        throw Exception("CgWangTilesT: illegal tile mapping texture size");
        }
    }

    // corner mapping texture size
    int cornerMappingTextureHeight = 0;
    int cornerMappingTextureWidth = 0;

    if(cornerMappingTextureID > 0)
    {
        glActiveTextureARB(GL_TEXTURE1_ARB);
        glBindTexture(GL_TEXTURE_RECTANGLE_NV, cornerMappingTextureID);

        glGetTexLevelParameteriv(GL_TEXTURE_RECTANGLE_NV, 0,
            GL_TEXTURE_HEIGHT, &cornerMappingTextureHeight);
        glGetTexLevelParameteriv(GL_TEXTURE_RECTANGLE_NV, 0,
            GL_TEXTURE_WIDTH, &cornerMappingTextureWidth);

        if( (cornerMappingTextureHeight <= 0) || (cornerMappingTextureWidth <= 0) )
        {
            throw Exception("CgWangTilesT: illegal corner mapping texture size");
        }
    }

```

```
if((tileMappingTextureHeight &
        ornerMappingTextureHeight &
        (tileMappingTextureHeight != cornerMappingTextureHeight)) ||
        (tileMappingTextureWidth &
        cornerMappingTextureWidth &
        (tileMappingTextureWidth != cornerMappingTextureWidth))
)
{
        throw Exception("CgWangTilesT: incompatible tile and corner mapping texturesizes");
}

string program = \
        CreateProgram(tileHeight, tileWidth,
                tileTextureHeight, tileTextureWidth,
                cornerTextureHeight, cornerTextureWidth,
                tileMappingTextureHeight, tileMappingTextureWidth,
                cornerSharpness);

_program = cgCreateProgram(_context, CG_SOURCE, program.c_str(), _fragmentProfile,"fragment", 0);

#if 0
        cout << program << endl;
        cout<<"---- PROGRAM BEGIN ----"<<endl;
        cout<<cgGetProgramString(_program, CG_COMPILED_PROGRAM);
        cout<<"---- PROGRAM END ----"<<endl;
#endif

if(_program)
{
        cgGLLoadProgram(_program);

        CheckError();
}

// tile texture
if(tilesTextureID > 0)
{
        glActiveTextureARB(GL_TEXTURE0_ARB);
        _tilesTexture = cgGetNamedParameter(_program, "tilesTexture");
        cgGLSetTextureParameter(_tilesTexture, tilesTextureID);
        cgGLEnableTextureParameter(_tilesTexture);
}

// corner texture
if(cornersTextureID > 0)
{
glActiveTextureARB(GL_TEXTURE1_ARB);
_cornersTexture = cgGetNamedParameter(_program, "cornersTexture");
cgGLSetTextureParameter(_cornersTexture, cornersTextureID);
cgGLEnableTextureParameter(_cornersTexture);
}

// tile mapping texture
if(tileMappingTextureID > 0)
{
_tileMappingTexture = cgGetNamedParameter(_program, "tileMappingTexture");
cgGLSetTextureParameter(_tileMappingTexture, tileMappingTextureID);

cgGLEnableTextureParameter(_tileMappingTexture);
}
```

```

    // corner mapping texture
    if(cornerMappingTextureID > 0)
    {
        _cornerMappingTexture = cgGetNamedParameter(_program, "cornerMappingTexture");
        cgGLSetTextureParameter(_cornerMappingTexture, cornerMappingTextureID);

        cgGLEnableTextureParameter(_cornerMappingTexture);
    }

    // enable stuff
    cgGLEnableProfile(_fragmentProfile);

    cgGLBindProgram(_program);
}

CgWangTilesT::~CgWangTilesT(void)
{
    Disable();
    cgDestroyContext(_context);
}

void CgWangTilesT::CheckError(void)
{
    CGerror error = cgGetError();

    if(error != CG_NO_ERROR)
    {
        throw Exception(cgGetErrorString(error));
    }
}

void CgWangTilesT::Enable(void) const
{
    if(_tileMappingTexture) cgGLEnableTextureParameter(_tileMappingTexture);
    if(_cornerMappingTexture) cgGLEnableTextureParameter(_cornerMappingTexture);
    if(_tilesTexture) cgGLEnableTextureParameter(_tilesTexture);
    if(_cornersTexture) cgGLEnableTextureParameter(_cornersTexture);

    cgGLEnableProfile(_fragmentProfile);

    cgGLBindProgram(_program);
}

void CgWangTilesT::Disable(void) const
{
    if(_tileMappingTexture) cgGLDisableTextureParameter(_tileMappingTexture);
    if(_cornerMappingTexture) cgGLDisableTextureParameter(_cornerMappingTexture);
    if(_tilesTexture) cgGLDisableTextureParameter(_tilesTexture);
    if(_cornersTexture) cgGLDisableTextureParameter(_cornersTexture);

    cgGLDisableProfile(_fragmentProfile);
}

string CgWangTilesT::CreateProgram(const int tileHeight,
        const int tileWidth,
        const int tileTextureHeight,
        const int tileTextureWidth,
        const int cornerTextureHeight,
        const int cornerTextureWidth,
        const int tileMappingTextureHeight,
        const int tileMappingTextureWidth,
        const float cornerSharpness)
{
  strstream strResult;

  strResult << "struct FragmentInput" << endl;
  strResult << "{" << endl;
  strResult << " float4 tex : TEX0;" << endl;
  strResult << " float4 col : COL0;" << endl;
  strResult << "};" << endl;
  strResult << "struct FragmentOutput" << endl;
  strResult << "{" << endl;
  strResult << " float4 col : COL;" << endl;
  strResult << "};" << endl;
  strResult << "" << endl;
  strResult << "float2 mod(const float2 a, const float2 b)" << endl;
  strResult << "{" << endl;
  strResult << " return floor(frac(a/b)*b);" << endl;
  strResult << "}" << endl;
  strResult << "" << endl;
  strResult << "FragmentOutput fragment(FragmentInput input," << endl;
  strResult << " uniform sampler2D tilesTexture," << endl;
  strResult << " uniform sampler2D cornersTexture," << endl;
  strResult << " uniform samplerRECT tileMappingTexture," << endl;
  strResult << " uniform samplerRECT cornerMappingTexture)" << endl;
  strResult << "{" << endl;
  strResult << " FragmentOutput output;" << endl;

  strResult << " float2 mappingScale = float2(" <<
tileMappingTextureWidth << ",
" << tileMappingTextureHeight << ");" << endl;
  strResult << " float2 mappingAddress = input.tex.xy * mappingScale;"
<<endl;

  if( (tileTextureHeight > 0) && (tileTextureWidth > 0))
  {
        strResult << " float2 tileScale = float2(" << tileTextureWidth
<< ", " << tileTextureHeight << ");" << endl;
        strResult << " float2 tileScaledTex = input.tex.xy * float2("
<< tileMappingTextureWidth*1.0/tileTextureWidth << ", " <<
tileMappingTextureHeight*1.0/tileTextureHeight << ");" << endl;
  strResult << " float4 whichTile = texRECT(tileMappingTexture, mod
(mappingAddress, mappingScale));" << endl;
  strResult << " float4 result1 = tex2D(tilesTexture, (whichTile.xy +
frac(mappingAddress))/tileScale, ddx(tileScaledTex), ddy(tileScaledTex));" <<
endl;

  strResult << " output.col =
lerp(result1,float4(0.207,0.368,0.549,1.0),whichTile.a);" << endl;
  }

  if( (cornerTextureHeight > 0) && (cornerTextureWidth > 0))
  {
    strResult << " float2 cornerScale = float2(" << cornerTextureWidth
<< ", " << cornerTextureHeight << ");" << endl;
    strResult << " float2 cornerScaledTex = input.tex.xy * float2(" <<
    tileMappingTextureWidth*1.0/cornerTextureWidth << ", " <<
    tileMappingTextureHeight*1.0/cornerTextureHeight << ");" << endl;
    strResult << " mappingAddress.xy = mappingAddress.xy - float2(0.5, 0.5);" << endl;
    strResult << " float4 whichCornerTile = texRECT(cornerMappingTexture, mod(mappingAddress, mappingScale));" << endl;
    strResult << " float4 result2 = tex2D(cornersTexture, (whichCornerTile.xy + frac(mappingAddress) + float2(0.5, 0.5))/cornerScale, ddx(cornerScaledTex), ddy(cornerScaledTex));" << endl;
    strResult << " output.col = result2;" << endl;
    }

    if( (tileTextureHeight > 0) && (tileTextureWidth > 0) &&
    (cornerTextureHeight > 0) && (cornerTextureWidth > 0) )
    {
    strResult << " float2 cornerDistance = frac(mappingAddress) - float2(0.5, 0.5);" << endl;
    strResult << " float bweight = exp(-dot(cornerDistance, cornerDistance)/" << cornerSharpness << ");" << endl;
    strResult << " output.col = result1*(1 - bweight) + result2*bweight;" <<endl;

    }

    strResult << " return output;" << endl;
    strResult << "}" << endl;

    // done
    string result(strResult.str(), strResult.pcount());
    strResult.rdbuf()->freeze(0);
    return result;
}

```