TP012: Terrain Paging Using Multiple Threads

Introduction

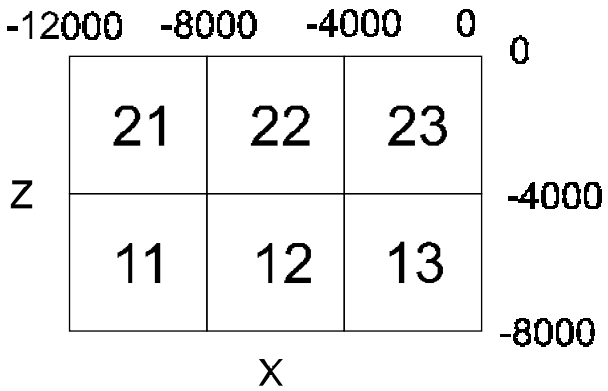
A scene which contains a large terrain can needlessly hinder the time to render a view since only a small part of the total terrain may be visible. It is advantageous to determine which regions of the terrain are visible and only have these in memory. This increases rendering performance by reducing the amount of geometry considered for rendering and optimises memory usage. In this paper terrain visibility is determined using a simple test based upon where the current camera is located.

This document shows the use of a separate thread of execution to control the visibility of a terrain model by dynamically paging in and paging out sections of the terrain from memory. This paper extends the application created by the RealiMation AppWizard (as discussed in *Technical Paper 003 : Using RealiMation in Microsoft Windows Applications*) and assumes a basic understanding of multi-threading within Microsoft Windows. It would also be an advantage to have completed the application enhancements discussed in *Technical Paper 004 : Your First RealiMation Windows™ Application* to provide a controllable object which can be moved within the standard Helisim RealiBase.

Preparing the RealiBase

The basic approach taken in this paper is to have a single main RealiBase which is loaded by the viewer application. Separate RealiBase files are used to store individual sections of the terrain and these will be loaded on demand by the application. On loading a new part of the terrain, geometry will be added and removed from the main RealiBase held in memory.

The standard Helisim RealiBase (provided with the RealiMation package) contains a terrain model which is comprised of six landscape tiles. These tiles are of approximately equal dimension and are named “11”, “12”, “13”, “21”, “22” and “23”. These tiles are oriented in the XZ plane as the following diagram shows:



The first step in creating a set of RealiBases for terrain paging is to create separate RealiBase files for each of the terrain tiles. Within each of these will be held the necessary materials and images for the tile, the geometry for the tile of interest and empty shapes for those that are to be removed from the scene. The procedure for achieving this is outlined below:

1. Take a copy of the “`Helisim.RBS`” RealiBase and open it in the STE.
2. In the Shapes lister select the terrain geometries “11”, “12”, “13”, “21”, “22”, “23” and select Optimise|Fix Vertex Colours. This step is necessary to preserve the pre-lighting of the terrain shapes which will be lost when the view is removed below.
3. Delete all Views, Actions, Camera, Placements, Atmospherics and Lights ignoring any warnings regarding references. Also delete all Shapes except “11”, “12”, “13”, “21”, “22”, “23”, all Materials except “Light grass” and “Road”, and all Images except “roadsurf.bmp” and “ground.bmp”. Save this version of the RealiBase to a new temporary file.

4. Take six copies of the RealiBase created in Step 3 and name them "zone_11.RBS", "zone_12.RBS", "zone_13.RBS", "zone_21.RBS", "zone_22.RBS", and "zone_23.RBS" respectively.
5. For each of the RealiBases created in Step 4 open it in the STE and delete all shapes except the zone of interest and create empty shapes of the same name for those that have just been removed. For example, edit "zone_21.RBS" and delete all shapes except "21". Now create five empty shapes and call them "11, "12", "13", "22" and "23" respectively.

Pre-prepared versions of the above RealiBases are provided with this documentation. Note that the original Helisim RealiBase remains unchanged.

Outline of the Approach

In this simple application only one of the terrain tiles will be loaded for any camera position with the other tiles being deleted from the RealiBase held in memory. This will result in a decrease in rendering time and memory usage due to the simplification of the scene. However two issues must be resolved regarding monitoring where the current camera is and the loading of the relevant terrain tile, if necessary. Two basic approaches can be used:

1. Insert a check in the rendering loop to see if the camera has moved into a new zone in the scene. If it has then load in the new tile of the terrain and remove the unused tile. This, however, requires that the render loop be interrupted whilst the camera position is inquired and also during the process of loading the new tile from a file.
2. Use a separate thread of execution, with a low priority, to monitor the current camera position and load a new terrain tile if required. This can occur simultaneously with the render loop and with the minimal of interruption. Provided the terrain paging thread is given a low priority the rendering speed should not be adversely affected. There will, however, be some processing overhead with the inclusion of the new thread and the subsequent file I/O operations. The extent of these is operating system and hardware dependent.

Loading a new tile and removal of the unwanted geometry information can be achieved using a single call to `RTLoadRealiBase()` given the data files that where created above. Setting the flags `RTReplaceMatchedNames` and `RTDeleteMatchedNames` ensures that when one of the terrain RealiBases is loaded any currently loaded shapes with the same names are deleted and replaced by the incoming ones. This explains why empty geometries have been created for the unwanted shapes in each of the terrain RealiBases. To avoid unnecessary loading of the materials and images for each of the terrain tiles the flag `RTMatchMaterialsOnLoad` should be set. Since all the materials and images are loaded when the main RealiBase is accessed on File|Open it is not necessary to load them again when a new tile is accessed. They are included so that references to material properties and images are maintained.

Determining which zone the camera is currently in is a simple case of inquiring the camera information and comparing the position co-ordinates with the extent of each terrain tile.

Adding a CTerrainControl Class

This section assumes that either a new application has been created using the RealiMation AppWizard or that you are expanding upon the application developed in *Technical Paper 004 : Your First RealiMation Windows™ Application*.

First insert a new generic class called `CTerrainControl`. This will contain member functions for the creation of a new thread and the terrain loading. It will also hold flags to control the creation and killing of threads. These are used to ensure that the application is aware of the state of each thread and to indicate when a thread is to terminate. These flags are also used to block the parent process whilst the thread is started and during is termination. These blocks ensure a consistent state of variables between parent and child process and is good practice when dealing with multiple threads. Note that a return from a call to

`AfxBeginThread()` does not imply that the thread has started, only that it is scheduled to start.

The flags used to control the life of the thread, `m_stop` and `m_running`, are declared `volatile` since they may be set by differing processes.

The declaration of the `CTerrainControl` class is:

```
////////////////////////////////////////////////////////////////////
// RealiBase viewer application
// Copyright (c) Datapth Ltd. 1998
//
// TerrainControl.h: interface for the CTerrainControl class.
//
////////////////////////////////////////////////////////////////////

class CTerrainControl
{
public:
    CTerrainControl();
    virtual ~CTerrainControl();

    void Initialise(CString&);
    void UnInitialise();

    void StartTerrainPaging();
    void StopTerrainPaging();

private:
    static CString m_directory;       // Path to find terrain RealiBases.

    static BOOL volatile m_stop;      // Flag to indicate that thread is to die.
    static BOOL volatile m_running;   // Flag to indicate that thread is active.

    static UINT TerrainPagingControl(LPVOID param);
};
```

The complete class definition is provided in Appendix A but is outlined below:

- `m_stop` and `m_running` private member flags. These allow the parent process to know and control the state of the child thread. If `m_running` is true the thread is active. If `m_stop` is true the child thread is scheduled to die.
- `Initialise()` and `UnInitialise()` member functions. In this simple application these simply provide a public interface to stop, start and restart the terrain control thread. The string provided to `Initialise()` gives the path in which to find the terrain RealiBases. This path, which is stored in `m_directory`, is taken to be the same directory in which the main RealiBase is located. In a more complex application these functions could be used for other initialisation functions.
- `StartTerrainPaging()` and `StopTerrainPaging()` member functions. These provide a public interface for explicit starting and stopping of the terrain paging thread. Note that these functions wait on the member flags for a successful start or stop of the thread. Such blocking waits ensure consistency between the state of the thread and the member flags:

```
void CTerrainControl::StartTerrainPaging()
{
    if ( !m_running )
    {
        m_stop = FALSE;
         AfxBeginThread(TerrainPagingControl, NULL, THREAD_PRIORITY_IDLE);
        while ( !m_running );              // Wait for thread to start.
    }
}

void CTerrainControl::StopTerrainPaging()
{
    if ( m_running )
    {
        m_stop = TRUE;
        while ( m_running );               // Wait for thread to die.
    }
}
```

- `TerrainPagingControl()` private member function. This function is the main control of the

thread. Once called by `AfxBeginThread()` the thread will survive for the duration of this function, hence the main `while()` loop. Once inside the loop the thread is active until the `m_stop` flag is set by the parent process. The main activity of the thread is to monitor the current position of the camera, determine which of the six zones it is in and ensure that the correct zone RealiBase is loaded. A simplified, pseudo-code, version of this function is shown below:

```
UINT CTerrainControl::TerrainPagingControl(LPVOID param)
{
    // First action is to indicate to the parent that the thread is active.
    m_running = TRUE;

    // This is the main body of the thread which is executed until
    // the stop flag is set by the parent thread.
    while ( !m_stop )
    {
        // Get the current camera position and determine which of the
        // six zones it is above. This gives the name of the terrian
        // RealiBase to load.

        if ( camera is in a new zone )
        {
             RTLoadRealiBase(current_zone, RTMatchMaterialsOnLoad |
                                           RTDeleteMatchedNames |
                                           RTReplaceMatchedNames);
        }
    }

    // Indicate to the parent that this thread is no longer running
    m_running = FALSE;

    return 0;
}
```

Activating the Terrain Paging

We wish for the terrain paging control thread to start once the main Helisim RealiBase is loaded and to terminate on exiting the application or on loading a new file. A single, global, `CTerrainControl` object is declared in the "`TerrainControl.cpp`" file. This `theTerrainController` object is the only instance of the thread code.

On loading a new RealiBase, in `LoadRealiBase()`, the thread paging is stoped prior to the load by calling `theTerrainController.UnInitialise()` and started by calling `theTerrainController.Initialise()` after the load. The path from which the main RealiBase is loaded is passed to the `Initialise()` function. On exiting the application, in `ExitInstance()`, the terrain paging is stopped by calling `theTerrainController.UnInitialise()`.

Implementation Issues

The approach used in this application has been simplistic and would need expanding upon to give more satisfactory results. By creating a new camera with a circular path to orbit the centre of the whole terrain it can be seen that the application does successfully page in and page out the terrain tiles based on camera location. There are some problems however:

- There is a sudden and very visible swap between one tile and another as the camera passes over a boundary. By loading a neighbourhood of tiles, or pre-empting the trajectory of the camera, the swapping could be made less visible.
- The code makes no attempt to determine where the camera is looking. Firing rays through the view to determine which parts of the terrain are visible could help to load only those parts that are required.
- Since the thread has been a given the low priority of THREAD_PRIORITY_IDLE, there is the possibility of some delay between the camera entering a new zone and the relevant tile becoming visible. Increasing the priority to THREAD_PRIORITY_LOWEST would reduce this delay but, on single processor system, would increase the processor overhead of having the separate thread.

- If any paths have been set to “terrain follow” there will be problems if the elevation being followed is paged out. Care should be taken to avoid such circumstances.

Adding an extra thread to control the terrain paging as opposed to inserting code within the render loop does not give the programmer “something for nothing”. Resources are still going to be required to execute the new thread so there is always going to be an overhead. This can be significantly reduced on multiprocessor systems where the operating system performs load balancing. To reduce overhead, the thread has, in this case, been given a low priority. Given this, there is no guarantee of when the thread will be allocated resources by the operating system. If the application requires a deterministic response from the thread the programmer should consider reassigning the thread priority. Alternatively a single thread application should be used and the render loop interrupted at known opportune times.

Summary

This paper has outlined the use of multiple threads in a RealiMation application. This is made possible due to the thread safe nature of the RealiMation libraries. A separate thread of execution has been used in this instance to control the loading and removal of terrain data based on the current camera position. The simple application developed here, whilst being far from complete, serves to show the potential for such an approach. Methods for improving the performance of the application have been discussed but it should be noted that these would only increase resources required by the new thread.

Appendix A : CTerrainControl Object Code

This section contains the complete code for the `CTerrainControl` object. Paste it into a new source file called “`TerrainControl.cpp`”, and add it to your project.

```
////////////////////////////////////////////////////////////////////
// RealiBase viewer application
// Copyright (c) Datapth Ltd. 1998
//
// TerrainControl.cpp: implementation of the CTerrainControl class.
//
////////////////////////////////////////////////////////////////////

#include “stdafx.h”
#include “MainFrm.h”
#include “rmdebug.h”
#include “TerrainApp.h”
#include “Channel3D.h”
#include “TerrainControl.h”

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

/*————————————————————*/
// The one and only CTerrainControl object
CTerrainControl theTerrainController;

/*————————————————————*/
// Define static class members
BOOL volatile CTerrainControl::m_stop;
BOOL volatile CTerrainControl::m_running;

CString CTerrainControl::m_directory;

/*————————————————————*/
// Define array of RealiBase file names for each of the terrains
static CString terrains[6] = {“zone_11.rbs”, “zone_12.rbs”, “zone_13.rbs”,
                              “zone_21.rbs”, “zone_22.rbs”, “zone_23.rbs”};
```

```
////////////////////////////////////////////////////////////////////////
//  Construction/Destruction
////////////////////////////////////////////////////////////////////////

CTerrainControl::CTerrainControl()
{

}

CTerrainControl::~CTerrainControl()
{

}

/*_____________________________________*/
void  CTerrainControl::Initialise(CString&  dir)
{
    // Stop any paging that is currently active and restart.
    StopTerrainPaging();
    m_directory = dir;
    StartTerrainPaging();
}

/*_____________________________________*/
void  CTerrainControl::UnInitialise()
{
    // Stop any paging that is currently active.
    StopTerrainPaging();
}

/*_____________________________________*/
void  CTerrainControl::StartTerrainPaging()
{
    // Provided a terrain paging thread is not active, begin a new
    // thread and wait for a successful start.
    if ( !m_running )
    {
        m_stop = FALSE;
        // Execute thread with lowest priority so as to have minimal
        // affect on the parent rendering process.
         AfxBeginThread(TerrainPagingControl,  NULL,  THREAD_PRIORITY_IDLE);
        while ( !m_running );              // Wait for thread to start.
    }
}

/*_____________________________________*/
void  CTerrainControl::StopTerrainPaging()
{
    // If the terrain paging thread is running, set the stop flag and wait
    // for a successful stop.
    if ( m_running )
    {
        m_stop = TRUE;
        while ( m_running );               // Wait for thread to die.
    }
}

/*_____________________________________*/
UINT  CTerrainControl::TerrainPagingControl(LPVOID  param)
{
    // This is the main control for the terrain paging thread.
    // The thread survives for the life of this function hence the
    // use of static class members to act as flags. These indicate
    // when the thread is running and when it should die.

    // First action is to indicate to the parent that the thread is active.
    m_running = TRUE;

    // Variable initialisation that can be done outside the main loop.
    CString  *current_zone=NULL,  *loaded_zone=NULL;
    rtID camera_id;
    rtCamera camera;
    rtPoint3 *pos;
    CMainFrame *pFrame = (CMainFrame*) theApp.m_pMainWnd;   // Current frame.
    rtID view_id = pFrame->Get3DView().GetViewID();         // Current view ID.
```

```
    // This is the main body of the thread which is executed until
    // the stop flag is set by the parent thread.
    while ( !m_stop )
    {
        // Get the current camera position and determine which of the
        // six zones it is above. This gives the name of the terrian
        // RealiBase to load.

        camera_id = RTInqObject(view_id, RTCameraID);        // Current camera ID
        RTInqCamera(camera_id, &camera, NULL);               // Get camera position.
        pos = &camera.position.pos;

        // Terrain consists of six tiles of roughly equal area in the XZ plane
        // Terrain extent is from (0,0) to (-12000, -8000) with each terrain tile
        // having a dimension of 4000 by 4000.
        if (pos->z > -4000)
        {
            if (pos->x > -4000)
            {
                current_zone = &terrains[5];
            }
            else if (pos->x > -8000)
            {
                current_zone = &terrains[4];
            }
            else
            {
                current_zone = &terrains[3];
            }
        }
        else
        {
            if (pos->x > -4000)
            {
                current_zone = &terrains[2];
            }
            else if (pos->x > -8000)
            {
                current_zone = &terrains[1];
            }
            else
            {
                current_zone = &terrains[0];
            }
        }

         if (current_zone!=NULL && current_zone!=loaded_zone)
        {
            // Camera is in a new zone so load in the relevant RealiBase
            // file. Since each of the terrain files has empty geometries
            // for the 'unseen' terrains there is no need to explicitly
            // delete them from the current RealiBase in memory.
            CString filename = (m_directory + *current_zone);

             RTLoadRealiBase(filename, RTMatchMaterialsOnLoad |
                                       RTDeleteMatchedNames |
                                       RTReplaceMatchedNames);

            loaded_zone = current_zone;
            RMTRACE1("Loaded terrain zone %s", filename);
        }
    }

    // Indicate to the parent that this thread is no longer running
    m_running = FALSE;

    return 0;
}
/*_____________________________________*/
```