TP004: Your First RealiMation Windows™ Application

Introduction

This document assumes a familiarity with C++ programming techniques, Microsoft Developer Studio and ClassWizard. the RealiMation STE and the basic RealiMation Windows application discussed in Technical Paper no. 003 ("Using RealiMation in Microsoft Windows™ Applications").

This document will lead you through the process of enabling the user to control the motion of objects within a RealiBase by using a joystick.

The steps described are:

1. Creating the basic application.
2. Taking control of an object.
3. Making the object move via program calls.
4. Making the object move under joystick control.
5. Improving the usability of the application.

Creating the Application

First you will need to create the basic RealiMation application. To do this run up Microsoft Developer Studio and create a new project based on the "RealiMation AppWizard". Enter "TankApp" as the name of the application.

If you don't have this as a choice see the file "RealiMation AppWizard.txt" in the SDK directory. This document explains how to install the wizards for your compiler.

Once the AppWizard has completed, build the executable. It is recommended that the Debug build is used for the rest of this tutorial. Select this from Developer Studio from the 'Build | Set Active Configuration' menu option, and choose "TankApp - Win32 Debug".

When you try and compile up the debug version, your compiler will show an error in TankApp.cpp. This error is generated from the source code with a `#error` directive to warn you that you are compiling a debug version of your application without using the debug version of the RealiMation libraries. To remove this error, you should add "`_RMDEBUG`" to you compiler preprocessor definitions for the debug build only. To do this select 'Project | Settings' from the main menu and click on the C/C++ tab. Ensure that "Win32 Debug" is the selected target then add "`,_RMDEBUG`" to the end of the preprocessor definitions section.

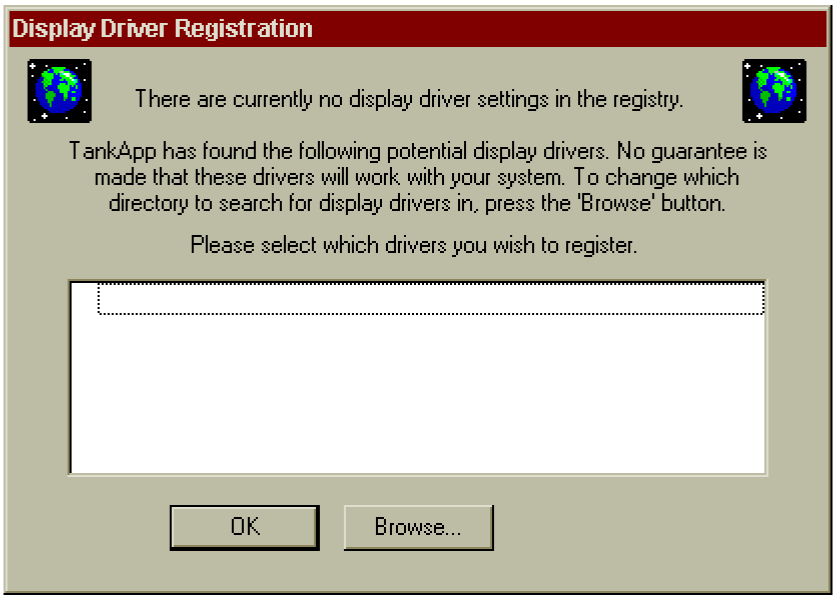
Alternatively, if you do not want to take advantage of the debugging facilities provided by RealiMation, simply comment out the `#error` line in the code.

Rebuild the application and run it.

The first thing that happens, assuming you are running the `_RMDEBUG` version, is that you get asked whether to turn on memory tracing or not. This is a feature enabled by the debug libraries (but controlled from the application) that helps track down memory usage and leaks within any RealiMation components that you may be using. See the `CTankApp::InitInstance()` method for more details. Select "No" to the question.

The main application window then appears, with a large cross in the display area and a message saying "No Current RealiBase". Go to the 'File' menu, and select 'Open'. This prompts for a RealiBase file to use. You can load any file, but for now load "Helisim2.rbs", which is in the "Samples\Helisim" directory of your RealiMation CD.

On selection of the file, your application displays the following dialog:


Display Driver Registration

There are currently no display driver settings in the registry.

TankApp has found the following potential display drivers. No guarantee is made that these drivers will work with your system. To change which directory to search for display drivers in, press the 'Browse' button.

Please select which drivers you wish to register.

Since this is the first time that the tank application has been run, it does not have any display drivers set up for it, and so asks the user. Press the ‘Browse’ button, and select the “Bin” directory under your main RealiMation installation. The application then examines all files in this directory, determining if they are display drivers or not. The resulting list displayed in the above dialog shows all the drivers it has found. Select all those drivers you may wish to use in the future by clicking in the checkbox to the left of each entry. You have now registered with the application all the drivers you may want to dynamically swap between in the future. See the file `Driver.cpp` for more details.

When you hit OK, the program asks you to choose a default driver from the list. Choose one that suits you (probably the same as you have been using for the STE). The ‘Alternate’ tab allows applications to have a driver that only works in a window, and provides an emergency escape back to the Windows desktop if you have been using a full screen driver as the primary device. See source code for more details.

Please note that all this driver setup will only happen the first time you run the application. If, when you first run your application, the above dialogs do not happen, then it simply means that you may have run a RealiMation custom AppWizard program before, and the registry entries are being reused.

After you select OK, a picture should appear. You can then use the menu options to control the scene in a few simple ways, such as changing the camera, animating the scene, and switching display mode and driver.

Once you have built the application as described, the code examples given below can be typed or pasted into your source.

Taking Control

To take control of an object we need to select the instance (“placement” in STE terms) and store its ID. We also need to remove any paths (“actions” in STE terms) attached to the instance so that we can move it anywhere independently. However, any actions on sub-placements are left intact so that, for example, the turret still swings from side to side.

Preparing a RealiBase to Use

This tutorial will use a copy of an existing RealiBase. Run the STE and load "Helisim2.rbs" as described above. We will make some changes to this and save it to your hard disk under a different name to be used by the tank application.

From within the STE, rename the placements "Target1" to be "Tank 1. Then drop the "Free camera" onto the view and position it so that you can see the this tank. Ideally this should be just behind and above the tank we wish to control.

To save the RealiBase, we have two choices. Because the original is on a CD, we cannot write back to it. The most obvious solution is to use the File | Save As option, and save the file to your hard disk somewhere and call it "Control.rbs". The drawback of doing this is that the textures will still be stored on the CD, which means they may not be found when loading the RealiBase from hard disk. This is where the 'Deploy' operation comes in very useful, as it can take copies of all referenced textures and place them all in a central place along with the RealiBase file itself.

From the STE's 'File' menu, select 'Deploy'. Choose a directory on your hard disk - note that if the directory you type in does not exist, then it will optionally be created for you. When this operation is complete, all the images referenced by this RealiBase will be copied into that directory, along with the RealiBase file itself, which is still called "Helisim2.rbs". Use the 'File | Save As' command to save the RealiBase as "Control.rbs" in the same directory.

Now exit the STE, and run the tank application again, but use the new "Control.rbs" file. Once you have loaded this, it is added to the recent file list on the File menu, which will make testing easier as the tutorial progresses.

Using the RealiBase

Now we have created a RealiBase, we can add code to take control of named objects. This entails getting hold of a handle to the object (the "ID"), and removing any predefined paths that the instance may be using.

Using Microsoft Developer Studio add a menu option called "Grab Tank 1", and use ClassWizard to add a response function to the `CTankApp` class. Edit this code to return the ID of "Tank 1" using the call:

```
    RTGetIDFromName(“Tank 1”);
```

Then remove any existing path by calling:

```
    RTSetObject(tank_id, RTNullID, RTPathID);
```

(*Note*: this will work even if the object has no existing path). Your code should look like this:

```
/*————————————————————————————————————*/
void  CTankApp::OnControlTankGrabTank1()
{
   // Control tank 1

   // Look in RBS for object named “Tank 1”
   rtID tank_id = RTGetIDFromName (“Tank 1”);

   if (tank_id != RTNullID)
   {
      // Remove path from instance
      RTSetObject (tank_id, RTNullID, RTPathID);
   }
}
```

Build and run this version of the application. Play the animation (F10 key), and select the new menu option to take control of "Tank 1". When you press play you will notice that the tank has apparently disappeared from the view! To find out what has happened we need to load the modified RealiBase into the STE, and use the STE's model query features to quickly examine the modified data. Therefore we need the application to save the RealiBase. You could either add a menu option to do this or add the call:

```
RTSaveRealiBase (“debug.rbs”, NULL, RTSaveAll);
```

into the code after removing the path.

Inserting calls to `RTSaveRealiBase()` into your code and then using the STE to view the results can be a useful tool while developing and debugging your application. In effect, it is using the STE as a “data debugger”.

In the STE, load the debug RealiBase, find the views lister and select “Tank 1”. Hitting F3 to fit the view to this object or the spacebar to show the properties page will show that the tank is now at the world origin (i.e. its position is x=y=z=0). The reason is that, when originally adding a path to an instance, that path determined the instance’s position in the world, which is why it disappeared when the path is removed . The solution is to evaluate the path at the current time and assign the resulting transformation to the instance. To do this you will need to call `RTEvaluatePath()` to return the position and scale of the object and then `RTSetInstance()` to apply these values to the instance.

Your code should now look something like this:

```
/*————————————————————————————————————*/
void CTankApp::OnControlTankGrabTank1()
{
   // Control tank 1

   // Look in RBS for object named “Tank 1”
   rtID tank_id = RTGetIDFromName (“Tank 1”);

   ASSERT (tank_id != RTNullID);

   // Unhook it from any predefined action
   rtID pathid = RTInqObject (tank_id, RTPathID);

   if (pathid != RTNullID)
   {
      // Get current position of instance, then assign it permanently
      // to the instance when the path has been removed.
      rtPosition pos;
      rtScale    scale;

      RTEvaluatePath (pathid, theAnimations.Time(), NULL, NULL, &pos, &scale);

      // Remove path from instance
      RTSetObject (tank_id, RTNullID, RTPathID);

      // Move the instance to the correct place in the scene
      RTSetInstance (tank_id, &pos, &scale);
   }
}
```

We can also add a handler for an UPDATE_COMMAND_UI event, using ClassWizard, and inserting the following code:

```
/*————————————————————————————————————*/
void CTankApp::OnUpdateControlTankGrabTank1(CCmdUI* pCmdUI)
{
   // Disable option if no tank 1 object
   BOOL bEnable = (RTGetIDFromName (“Tank 1”) != RTNullID);
   pCmdUI->Enable (bEnable);
}
```

If you build and run this version you will see that the tank stops moving along the road at its current location when you select the menu option. The “`OnUpdate`” method simply greys out the menu option if there is no object called “Tank 1” in the current RealiBase. Supplying this method means that we can change the

```
   if (tank_id != RTNullID)
```

line to an: ASSERT (tank_id != RTNullID);

Since the user interface should never allow the grab tank command unless there is an object in the RealiBase of the right name, using an assertion can be a big debugging aid, and one which will not get compiled into a release build.

Adding Animation

The first task is to make the object spin. This illustrates animating an object under program control and acts as a check that we are doing the right steps to grabbing objects and manipulating them.

For the purposes of this tutorial we will change the `CAnimate` class to give it knowledge of an object to be controlled, and ultimately to read joystick input. In practice, your own applications may create a separate control class, or find some other place in the code to implement this functionality.

The way to do this is to add a public method into the `CAnimate` class.

```
/*————————————————————————————————————————*/
void  CAnimate::SetControlledObject  (rtID  id)
{
    // Sets the ID to be controlled every tick by a joystick
    m_JoyObject = id;
}
```

Where `m_JoyObject` is a protected member variable of type `rtID`. Add this to the `CAnimate` class declaration (in `animate.h`), and initialise its value to `RTNullID` in the `CAnimate` constructor.

Add a call to this method in `CTankApp::OnControlTankGrabTank1()`, which can be inserted after the `ASSERT` statement.

```
    ASSERT (tank_id != RTNullID);
    // Store this ID
     theAnimations.SetControlledObject  (tank_id);
```

Now that we have the object ID we can make it spin, via a new `CAnimate` method called `ControlObject()`.

In the method `CAnimate::Tick (void)` add the lines:

```
    // Update controlled object
    ControlObject();
```

just before the call to `pview->DrawView()`. Once this method has been written it will cause the position of the controlled object to be updated at each animation interval. Create a protected method called `ControlObject` to the `CAnimate` class, and add the following code to implement it:

```
/*————————————————————————————————————————*/
void  CAnimate::ControlObject  (void)
{
    // Check that there is an object to control
    if (m_JoyObject != RTNullID)
    {
        rtPosition pos;

        // Read the current position of object
        RTInqInstance (m_JoyObject, &pos, NULL);

        // Update its position
        pos.yaw += RMDegToRad (5.0f);
        RMReduceAngle (pos.yaw);  // Ensure in range 0 - 360 degrees

        // Write the changes back
        RTSetInstance (m_JoyObject, &pos, NULL);
    }
}
```

This function checks that an instance to control has been set. If an instance has been defined, its position is inquired, the yaw is incremented by 5 degrees and given back to the instance.

If you build and run this version you will see the tank stop moving along the road and start to spin when you select the menu option.

The application still doesn't do anything useful, but we have illustrated how easy it is to make an object do something under application control rather than moving along a predefined path. The next step is to add joystick control.

Adding Joystick Control

All the changes are now made in the `CAnimate` methods. For the purposes of this tutorial the Windows multimedia API's are used to access a joystick attached to your machine. Developers can use their own input gathering mechanisms if they prefer.

You will need to add the following as protected member variables to `CAnimate` to store information about the joystick:

```
JOYCAPS m_JoyCaps;
JOYINFOEX m_JoyInfo;
```

Then add the following lines to `CAnimate::SetControlledObject()` to initialise the joystick:

```
// Read joystick capabilities and ranges
m_JoyInfo.dwSize  = sizeof (JOYINFOEX);
m_JoyInfo.dwFlags = JOY_RETURNALL;

joyGetDevCaps (JOYSTICKID1, &m_JoyCaps,  sizeof (m_JoyCaps));
```

And change `CAnimate::ControlObject()` to look like this:

```
/*————————————————————————————————————————*/
void CAnimate::ControlObject()
{
   // Check that there is an object to control
   if (m_JoyObject != RTNullID)
   {
      // Read Position of object, modify with joystick
      rtPosition pos;

      RTInqInstance (m_JoyObject, &pos, NULL);

      // Handles joystick events
      MMRESULT result = joyGetPosEx (JOYSTICKID1, & m_JoyInfo);

      if (result != JOYERR_NOERROR)
      {
         return;
      }

      // Scale X & Y to +/- 10 Ord value
      Ord down;
      Ord across;

      across = 20.0F * (Ord)(m_JoyInfo.dwXpos - m_JoyCaps.wXmin) /
                       (Ord)(m_JoyCaps.wXmax  - m_JoyCaps.wXmin) - 10.0F;
      down   = 20.0F * (Ord)(m_JoyInfo.dwYpos - m_JoyCaps.wYmin) /
                       (Ord)(m_JoyCaps.wYmax  - m_JoyCaps.wYmin) - 10.0F;

      // Update the position of the object
      pos.pos.x += across;
      pos.pos.z -= down;

      // Write the changes back
      RTSetInstance (m_JoyObject, &pos, NULL);
   }
}
```

You will need to add the following code to "stdafx.h" in order to get the above code to compile:

```
#include <mmsystem.h>
```

and insert "winmm.lib" in the 'Object/library modules' field on the 'Project | Settings | Link' dialog to get the application to link.

Joysticks only return positive values, so to be able to move left and backwards the input has to be shifted into negative coordinates, hence the "- 10.0F" when calculating the joystick position.

If you build and run this version you will see that the tank moves a large distance for a relatively small movement of the joystick - you can easily lose sight of the tank. You can either scale the movement down by changing the hard coded values (20.0 and 10.0) or by inquiring the size of the object and using that to modify the numbers. It will also be useful to add a "dead zone" to the joystick movement around its central position, otherwise it is very difficult to stop the tank from moving

Using the bounding box of the controlled object to scale joystick movement is easy to do. Add the following member variable to the CAnimate class:

```
Ord m_JoyScaleFactor;
```

And the following code at the end of CAnimate::SetControlledObject() :

```
// Inquire bounding box of object
rtBox3d box;
RTInqObjectBox (m_JoyObject, 0.0f, &box);

rtVector diag;
RMBoxDiagonal (&box, &diag);
m_JoyScaleFactor = RMV3Length (&diag);
// Safety check
if (m_JoyScaleFactor < Epsilon)
   m_JoyScaleFactor = 1.0f;
```

This sets up a scale factor based on the diagonal length of the bounding box of the controlled object. The joystick reading code then changes to:

```
Ord  scale_offset = m_JoyScaleFactor * 0.5f;

across = m_JoyScaleFactor * (Ord)(m_JoyInfo.dwXpos - m_JoyCaps.wXmin) /
                 (Ord)(m_JoyCaps.wXmax  - m_JoyCaps.wXmin) - scale_offset;
down   = m_JoyScaleFactor * (Ord)(m_JoyInfo.dwYpos - m_JoyCaps.wYmin) /
                 (Ord)(m_JoyCaps.wYmax  - m_JoyCaps.wYmin) - scale_offset;
```

Finally, we can add a joystick dead zone to prevent movement when the joystick is around its centred state. For the purposes of this example, we simply filter out small changes in the across and down factors. This is done by changing the position update statements to:

```
// Update the position of the object
if (RMFabs (across) > (m_JoyScaleFactor * 0.1f))
   pos.pos.x += across;
if (RMFabs (down) > (m_JoyScaleFactor * 0.1f))
   pos.pos.z -= down;
```

If you now build and run this program, the tank should not move until the joystick is moved beyond a certain threshold.

You will also note that the camera remains at its initial location and the tank can very quickly disappear off the screen. Therefore we need to make sure we can always see the object we are controlling.

Usability Issues

Getting the camera to follow the object being controlled

You can do this in two ways, the first is to use the STE to make the "Free camera" always point at "Tank 1", by dragging the "Tank 1" instance and dropping it on "Free camera". This would be OK for this exercise in its current state, but not for the more general case in which a user could control any object, since we would like the camera to dynamically attach itself to whatever object the user selected to control.

The second solution is to change the code to make the camera always point at the object at the time the it is selected. To do this you will need to modify `CTankApp::OnControlTankGrabTank1()` to inquire the current camera and then call `RTSetAttachment()` to make that camera point at the tank. Add the following code to the end of the function:

```
CMainFrame *pFrame = (CMainFrame*) m_pMainWnd;
ASSERT (pFrame);
CChannel3D &channel = pFrame->Get3DView();
rtID view_id = channel.GetViewID();
rtID camera_id = RTInqObject (view_id, RTCameraID);

// RTSetAttachment requires a list of instances
rtID attach_list[2] = {tank_id, RTNullID};
RTSetAttachment (camera_id, attach_list, RTFollow);
```

This is the same code as the STE uses when a user drops an instance on a camera. The tank will now always be visible (unless it moves behind a hill or building in the scene!). Build and run the application to see the new functionality.

Getting the camera to move with the object being controlled

Instead of keeping the camera at its current position, and always pointing towards the moving vehicle, it would be nice to physically attach the camera to the tank so that the viewpoint appears to move with the vehicle.

To enable this, simply replace the final argument in the call to `RTSetAttachment()` with `RTMoveWith` in the `CTankApp::OnControlTankGrabTank1()` method. This is equivalent to dragging a camera and dropping it onto the instance from within the STE. This will, however, place the camera at the tank's local origin, so we also need to change the camera's offset above and behind the object to which it is attached. Insert the following code to move the camera backwards and upwards so that it can be seen, after calling `RTSetAttachment()`as above:

```
// Move camera back and up
rtPosition offset;
RTInqCamera (camera_id, NULL, &offset);
offset.pos.y += 3.0F;
offset.pos.z -= 5.0F;
RTSetCamera (camera_id, NULL, &offset);
```

Sine we are only setting the camera's offset, we can pass NULL for the 'position' parameters of `RTInqCamera()` and `RTSetCamera()`.

The camera will now move with the tank keeping a constant distance behind it all the time. Notice how the camera motion is purely a function of the linkage between the tank instance and the camera, freeing the application code from having to explicitly move both the tank and the camera.

Improving the motion dynamics

Currently, the tank can only move in directions parallel to the x and z axes of the world, and cannot alter the direction it is facing when the user takes control. This is hardly realistic, and makes driving in any direction which is not parallel to those axes (e.g., down the road) very difficult, or looking at objects in the scene from a different angle impossible.

The simple solution is to change the effect the joystick has on the tank. Instead of linking left/ right movement directly to the x-axis, we can link it to the yaw orientation of the tank :

```
// Update the position of the object
if (RMFabs (across) > (m_JoyScaleFactor * 0.1f))
{
   pos.yaw += RMDegToRad (across);
}
```

And instead of linking the forward/backward joystick movement to the z-axis, we can alter the position of the object based on the direction it is facing, allowing the tank to drive forwards and backward in whatever direction it is currently facing :

```
if (RMFabs (down) > (m_JoyScaleFactor * 0.1f))
{
   pos.pos.x -= RMSin (pos.yaw) * down;
   pos.pos.z -= RMCos (pos.yaw) * down;
}
```

This provides a much more useful motion dynamic. However, it does still suffer from one ‘feature’ - the tank moves every time the frame is drawn, and takes no account of how long that frame took to draw. This means that as the frame rate increases and decreases, then the apparent speed to the tank will also increase/decrease.

The solution is to link the amount the tank moves to the time since the last frame was drawn. Each time we move the tank, we can calculate the ‘time delta’ since the last time, and use this value as a scale factor.

To do this, we need to add a member variable `m_LastTime` to the `CAnimate` class definition. We will use this to hold the value of the time when the tank was last moved :

```
class CAnimate : public CObject

   ...

   rtID m_Terrain;      // Ground object for terrain following
   Ord  m_LastTime;     // For scaling object movement to time
```

We initialise this variable in the `CAnimate::SetControlledObject()` routine

```
m_Terrain = RTGetIDFromName (“Terrain”);

// Initialise the m_LastTime to ‘MaxFloat’, this means that
// the first time we compare the current time against the
// ‘last time’, we will always get a negative number
m_LastTime = MaxFloat;
```

Before we move the tank, we inquire the current time and calculate the difference from the `m_LastTime` value. By checking that this difference is positive, we can detect the first time the tank is moved, since the `m_LastTime` will be `MaxFloat`, making the difference negative :

```
across = m_JoyScaleFactor * (Ord)(m_JoyInfo.dwXpos - m_JoyCaps.wXmin) /
                  (Ord)(m_JoyCaps.wXmax  - m_JoyCaps.wXmin) - scale_offset;
down   = m_JoyScaleFactor * (Ord)(m_JoyInfo.dwYpos - m_JoyCaps.wYmin) /
                  (Ord)(m_JoyCaps.wYmax  - m_JoyCaps.wYmin) - scale_offset;

// Take account of how long it is since we drew the last frame -
//   the longer the time gap, the more we need to move.
// This means the tank will move at the same speed, taking the
// same amount of time to driver between two points, regardless of
// the frame rate

// Time factor - depends on how
Ord current_time     = RTInqClock();
```

```
    Ord distance_sfactor = 1.0F;
    Ord angle_sfactor    = 1.0F;

        // Amounts to scale the tank movement to
#define TANK_SPEED 30.0f
#define TURN_RATE  12.0f

    // If the current time is later than the last time,
    // then we scale the distance and angle...,
    if ((current_time - m_LastTime) >= 0.0F)
    {
        Ord time_delta = current_time - m_LastTime;
        distance_sfactor = time_delta * TANK_SPEED;
        angle_sfactor    = time_delta * TURN_RATE;
    }

    // Store the current time for next time round
    m_LastTime = current_time;

    // Update the position and orientation of the object...
    if (RMFabs (across) > (m_JoyScaleFactor * 0.1f))
    {
        // Rotate the object about the y-axis
        pos.yaw += RMDegToRad (across * angle_sfactor);
    }

    if (RMFabs (down) > (m_JoyScaleFactor * 0.1f))
    {
        // Move the object, depending on it's orientation
        pos.pos.x -= RMSin (pos.yaw) * down * distance_sfactor;
        pos.pos.z -= RMCos (pos.yaw) * down * distance_sfactor;
    }
```

Terrain following

Currently the tank can only move in its original plane, since the joystick code only changes the X and Z components of the tank's position. This means that it moves straight through any hills and flies over any valleys. The controlled object needs to be made to follow the terrain. This is done by using the `RTInqHeight()` function, which gets the height (y value) of geometry given the X and Z locations. All we need to do, therefore, is to find the object being used to represent the ground, and then get the elevation of the ground at our current tank position. This value is then used as the Y component of the object position.

To this end add a new protected member variable `m_Terrain` to `CAnimate` and the following code to `CAnimate::SetControlledObject()` (assuming that the ground object is called "Terrain" in the ReaIiBase):

```
    m_Terrain = RTGetIDFromName ("Terrain");
```

You may also want to initialise the new member variable to `RTNullID` in the constructor.

Then in `CAnimate::ControlObject()` add the following lines of code after the position of the object has been updated and before the call to `RTSetInstance()`:

```
        rtHeight ht;

        if (RTInqHeight (m_Terrain, &pos.pos, &ht))
        {
            pos.pos.y = ht.y;
        }
```

Build and run the application. When you driver the tank around, it will now climb slopes, and go up and down the hills. The tank stays horizontal, however, regardless of the slope of the land. Improving this is left as an exercise for the reader - *Hint:* You can use the face normal returned in the `rtHeight` structure and change the orientation of the tank to match.

Exercises

Let the user to select the object to control

As it stands the application can only control the object called “Tank 1”. This can be changed by modifying the code, but it would be nice to enable the user to select the object to control. The following are suggestions for implementing this feature:

- Change the menu option to present the user with a dialog into which the name can be typed or a picked from a list of instances.
- Let the user select the object using the mouse. *Hint:* See `RTRayTestView()`.

Further improving the motion model

We started with a crude movement model, allowing motion parallel to the x and z axes. We improved this by allowing the tank to turn, and move forwards and backwards. This model is still very simple, and a long way from ‘realistic’.

You might want to add acceleration and deceleration by reading how far the joystick has been moved, or simulate driving on different surfaces, with the tracks possibly slipping at different rates - the possibilities are endless.

The RealiGame source code sample implements other motion models, or you may already have your own that you can use.

Add Special Effects

The RealiMation Technical Note *TP009 : Creating Explosion Effects* with RealiMation, extends this tutorial to show how more dynamic effects (in this case explosions) can be implemented with the RealiMation API.

Summary

With only a few lines of code, we have added a terrain following joystick controlled tank that has a camera attached to it to the basic RealiMation AppWizard generated program. This application can easily be further extended to make even more use of the RealiMation API.