TP009: Creating Explosion Effects with RealiMation

Introduction

This tutorial can be done after *TP004: Your First RealiMation Windows Application* although the source code supplied shows the result of working through *TP004*, *TP010: 3D Sound in RealiMation* and then *TP009*. *TP010* finishes with a TankApp application in which you can take control of a tank and move it around the world accompanied by sound effects. This paper extends the application by adding routines to create an explosion effect using intersecting polygons with an alpha-mapped texture. The explosion will be animated by dynamically altering the scale and material of the geometry using the RealiMation API. The techniques demonstrated can also be applied to create many other kinds of special effect.

Familiarity is assumed with C++ programming techniques and using the RealiMation STE.

This document shows how to use the RealiMation API to:

- Create geometry using standard primitives such as `RTSphere()`
- Create geometry by setting up individual polygons with textured materials
- Attach geometry to instances
- Attach and detach instances to / from views
- Animate geometry over time
- Override the display mode for an object (e.g. to enable double-sided polygons)

Overview

A new menu item will be added to TankApp which when selected will cause the tank to explode. Initially the explosion will use a transparent glowing sphere which expands (adjusting the scale of the sphere) and then fades away (adjusting the transparency of the sphere's material) . After this we will show how the explosion's appearance can be improved by replacing the sphere with textured polygons. The texture used for the explosion will use an alpha map for per-pixel variable transparency.

Creating the Application

Before beginning this paper you should work through either *TP010: 3D Sound in RealiMation* or *TP004: Your First RealiMation Windows Application*. Alternatively if you are familiar with creating RealiMation applications you can begin with the source code supplied with either of these papers. You should now have the viewer application with a menu item which allows you to take control of the tank and move it with the joystick. Now we are going to add another menu item which will explode the tank.

First we need to create a new class to handle exploding objects - create new source and header files called `Explosion.cpp` and `Explosion.h` and add them to the TankApp project. The `CExplosion` class will have two methods - one to initiate an explosion of a given object at a given time and another to update the state of the explosion over time - this latter method will need to be called every frame.

The `Explosion.h` file should be entered as follows:

```
// Explosion.h - Declarations for the CExplosion class

#ifndef  __EXPLOSION_H__
#define  __EXPLOSION_H__

class  CExplosion
{
public:
    CExplosion ();
    ~CExplosion ();

    void CreateExplosion (rtID explodingObjectID, rtID viewID);
    void Update (void);
};

#endif
```

In order to explode “Tank 1” we will remove it from the view and replace it with explosion geometry when “Explode Tank 1” is selected from the menu. Initially we shall just remove the “Tank 1” instance from the view. This can be done using the `RTRemoveObject()` API call.

The `Explosion.cpp` file should be as follows:

```
// Explosion.cpp - Implementation of the CExplosion class

#include “stdafx.h”
#include “tankapp.h”
#include “channel3d.h”
#include “explosion.h”

CExplosion::CExplosion ()
{
}

CExplosion::~CExplosion ()
{
}

void CExplosion::CreateExplosion (rtID explodingObjectID, rtID viewID)
{
    // Remove the exploding object from the view
    RTRemoveObject (viewID, explodingObjectID);
}

void CExplosion::Update (void)
{
}
```

Now that we have an explosion class in place we need to reference it from the main application. First add a `CExplosion` object to the `CTankApp` class:

In `TankApp.h` insert

```
#include “explosion.h”
```

before the declaration of `class CTankApp : public CWinApp`, then add a public member variable

```
CExplosion m_Explosion;
```

to the `CTankApp` class.

Now use Microsoft Developer Studio to add a menu option called “Explode Tank 1”, and use the ClassWizard to add a response function to the `CTankApp` class called `OnControlTankExplodeTank1`. You should also add a handler for the UPDATE_COMMAND_UI event - this handler will be the same as that for `OnUpdateControlTankGrabtank1` so you just need to enter `OnUpdateControlTankGrabtank1` as the name and both events will share the same handler.

In `TankApp.cpp` the `OnControlTankExplodeTank1()` function needs to get the ID of the instance to be exploded - this can be done using the same `RTGetIDFromName()` call used in `OnControlTankGrabTank1()`. Alternatively you could store the instance ID selected in `OnControlTankGrabTank1()` and use that as the instance to explode - this would enable you to easily change the ‘target’ of the application.

In addition to the ID of the instance we are going to destroy we will also need the ID of the view containing the instance in order to call `RTRemoveObject()`. In TankApp the view ID is a member of `CChannel3D` which itself is a member of `CMainFrame`. In order to obtain the necessary IDs and then explode the tank your `OnControlTankExplodeTank1()` function becomes:

```
void CTankApp::OnControlTankExplodeTank1()
{
    // Explode Tank 1

    // Look in RBS for object named “Tank 1”
    rtID tankID = RTGetIDFromName (“Tank 1”);
    ASSERT (tankID != RTNullID);

    // Get the ID of the view being displayed
    CMainFrame *pFrame = (CMainFrame*) m_pMainWnd;
    ASSERT (pFrame);
    CChannel3D &channel = pFrame->Get3DView();
    rtID viewID = channel.GetViewID();

    // Explode the tank
    m_Explosion.CreateExplosion (tankID, viewID);
}
```

Now build and run the application. Start the view animating (F10) and then choose the ‘Explode Tank 1’ menu item. The tank should disappear from the view.

To get a close up view of the tank exploding you can select the ‘Grab Tank 1’ menu option before selecting ‘Explode Tank 1’. This camera may be too close to see the whole tank, if so you will need to edit the code in `CTankApp::OnControlTankGrabTank1()` to position the camera further behind the tank. Change the camera repositioning code to change the Z offset from

```
    offset.pos.z -= 5.0F;
```

to

```
    offset.pos.z -= 15.0F;
```

Now when you animate the view and select the menu item to grab the tank it will be clearly visible towards the centre of the screen.

Making the Tank Explode

Now you can see the tank disappearing from the view it’s time to add the explosion effect. In order to animate the explosion over time we need to call `Update()` for the explosion every frame. In `Animate.cpp` in `CAnimate::Tick()` after the call to `ControlObject()` add:

```
        // Update explosion
        theApp.m_Explosion.Update();
```

Now that we are calling `CExplosion::Update()` every frame we can add the code to animate the explosion.

In order to keep track of the current state of the explosion the `CExplosion` class will need the following extra variables which should be entered in `Explosion.h` as protected member variables:

```
    BOOL m_bExploding;          // flag true if explosion in progress
    Ord  m_StartTime;           // time at which the explosion began
    Ord  m_ExplosionScale;      // Size of explosion
    rtID m_ViewID;              // ID of view containing object
    rtID m_ExplosionGeomID;     // ID of explosion geometry
    rtID m_ExplosionInstID;     // ID of explosion instance
    rtID m_ExplosionMatID;      // ID of material for explosion
```

These variables should be initialised in the `CExplosion` constructor as follows:

```
CExplosion::CExplosion ()
{
    // Initialise member variables
    m_bExploding = False;
    m_StartTime = 0.0F;
    m_ExplosionScale = 0.0F;
    m_ViewID = RTNullID;
    m_ExplosionGeomID = RTNullID;
    m_ExplosionInstID = RTNullID;
    m_ExplosionMatID = RTNullID;
}
```

In addition to the new variables the explosion duration must be defined, as must the material used to colour the explosion. In `Explosion.cpp` after the `#include` lines add:

```
// Timing for the explosions
static const Ord sExpandTime = 0.6F;
static const Ord sFadeTime = 0.6F;
static const Ord sExplosionDuration = ( sExpandTime + sFadeTime );

// Initial material for explosion
static const rtMaterial sExplosionMaterial = {
     {0.0F, 0.0F, 0.0F},          // no reflective colour
     {1.F, 0.55F, 0.0F},          // emits an orange glow
     {0.0F, 0.0f, 0.0f, 0.0f},    // no reflective properties
     0.2f, 0                       // 20% transparent, no image associated
};
```

The static constants are used for the explosion timings. There will be two phases to the explosion, an expansion phase where the explosion simply grows in size and a fading phase where the explosions continues to grow but its material is made increasingly transparent until the explosion vanishes.

The `CExplosion::CreateExplosion()` function must be extended to create the geometry and material to use for the explosion and to add the geometry to the view by attaching it to an instance and placing that in the view. It is also necessary to find the correct position to place the explosion at. The revised `CreateExplosion()` function is below:

```
void CExplosion::CreateExplosion (rtID explodingObjectID, rtID viewID)
{
    m_bExploding = True;
    m_StartTime = RTInqViewTime(viewID);
    m_ViewID = viewID;

    // Create a material for the explosion using parameters from
    // sExplosionMaterial
    m_ExplosionMatID = RTCreateID(RTMaterialID);
    RTSetMaterial(m_ExplosionMatID, &sExplosionMaterial);

    // Create geometry for explosion:
    // Sphere, radius 0.5, full sphere, no capping, 12 segments, using
    // explosionMat
    m_ExplosionGeomID = RTCreateID(RTGeometryID);
    RTSphere(m_ExplosionGeomID, 0.5F, -PiBy2, +PiBy2, RTOff, RTOff, 12,
             m_ExplosionMatID);

    // Create an instance to reference the geometry
    m_ExplosionInstID = RTCreateID(RTInstanceID);

    // Add the geometry to the instance (at LoD 0)
    RTAddGeometryAtDetail (m_ExplosionInstID, 0, m_ExplosionGeomID);

    // Find the position of the explodingObject
    rtPosition pos;
    rtID pathid = RTInqObject (explodingObjectID, RTPathID);

    if (pathid != RTNullID)
    {
        // Find current position of an explodingObject on a path
        RTEvaluatePath (pathid, m_StartTime, NULL, NULL, &pos, NULL);
    }
    else
    {
        // Find current position of an explodingObject without a path
        RTInqInstance (explodingObjectID, &pos, NULL);
    }

    // Move the explosion to the correct position in the scene
    RTSetInstance (m_ExplosionInstID, &pos, NULL);

    // Set the scale of the explosion to 0
    rtScale scale = {0.0F, 0.0F, 0.0F};
    RTSetInstance (m_ExplosionInstID, NULL, &scale);

    // Set the scale of the explosion after sExpandTime seconds
    m_ExplosionScale = 8.0F;

    // Add the explosion instance to the view
    RTAddObject (viewID, m_ExplosionInstID);

    // Remove the exploding object instance from the view
    RTRemoveObject (viewID, explodingObjectID);
}
```

The value of 8.0 chosen for `m_ExplosionScale` is reasonable for the tank being exploded but if you were going to destroy different sized objects you would want the explosion size to be relative to the object size. The following code will adjust the scale of the explosion to be the same as the longest dimension of the object.

```
    // Calculate desired scale of explosion after sExpandTime seconds
    rtBox3d bBox;
    RTInqObjectBox (explodingObjectID, m_StartTime, &bBox);
    m_ExplosionScale = RMMax(bBox.max.x - bBox.min.x, bBox.max.y - bBox.min.y);
    m_ExplosionScale = RMMax(m_ExplosionScale, bBox.max.z - bBox.min.z);
```

Now that the explosion has been created and added to the view we need to add the `Update()` handler to adjust the scale and appearance of the explosion each frame. There are three separate cases to be handled depending on the time into the explosion:

Explosion expanding	time < `sExpandTime`
Explosion expanding and fading	`sExpandTime` < time < `sExplosionDuration`
Explosion finished - remove from view	time > `sExplosionDuration`

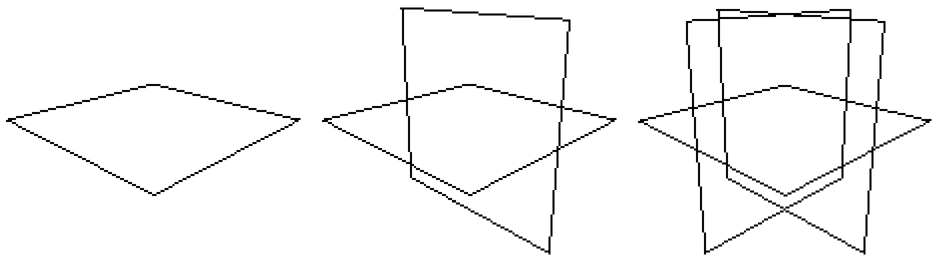
These cases can be seen in the new `CExplosion::Update()` function:

```
void CExplosion::Update (void)
{
    if (m_bExploding)
    {
        // Calculate how long the explosion has been animating
        Ord explosionTime = RTInqViewTime(m_ViewID) - m_StartTime;

        if (explosionTime < sExpandTime)
        {
            // Explosion still expanding
            Ord size = (explosionTime / sExpandTime) * m_ExplosionScale;

            // Set size of explosion
            rtScale scale = {size, size, size};
            RTSetInstance (m_ExplosionInstID, NULL, &scale);
        }
        else if (explosionTime < sExplosionDuration)
        {
            // Explosion fading
            Ord size = (explosionTime / sExpandTime) * m_ExplosionScale;

            // Set size of explosion
            rtScale scale = {size, size, size};
            RTSetInstance (m_ExplosionInstID, NULL, &scale);

            // Alter the material transparency to fade the explosion out
            rtMaterial tempMat;
            RTInqMaterial (m_ExplosionMatID, &tempMat);

            tempMat.transparency = sExplosionMaterial.transparency +
               ( ((explosionTime - sExpandTime) / sFadeTime) *
                 (1.0F - sExplosionMaterial.transparency) );

            RTSetMaterial (m_ExplosionMatID, &tempMat);
        }
        else if (explosionTime > sExplosionDuration)
        {
            // Explosion finished, remove it from the view & tidy up
            RTRemoveObject (m_ViewID, m_ExplosionInstID);
            m_bExploding = False;
            RTDeleteID (m_ExplosionInstID);
            RTDeleteID (m_ExplosionGeomID);
            RTDeleteID (m_ExplosionMatID);
        }
    }
}
```

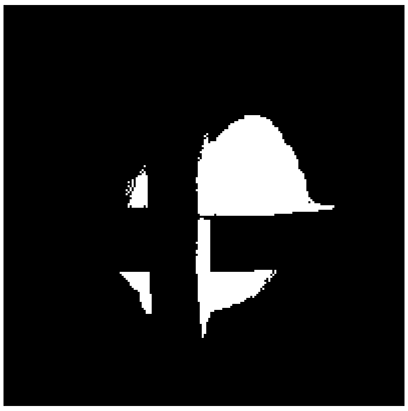
Now build and run the application. Once you have animated the view and then selected 'Explode Tank 1' you should see the tank replaced by a transparent orange globe which increases in size then fades away.

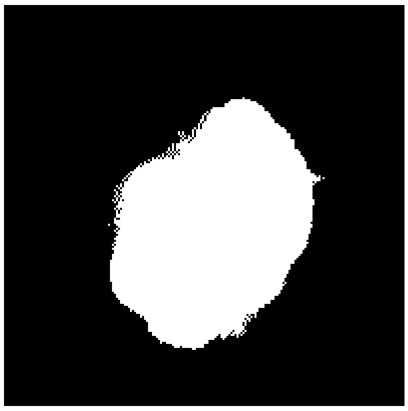
Enhancing the Appearance of the Explosions

We now have the explosion instance being added to the view at the location of the tank and an update routine that changes the scale and transparency of the explosion instance over time. In order to change the appearance of the explosion we just need to change the geometry and material used. For the revised explosions we are going to use three textured polygons that intersect at 90 degrees to each other in order to ensure the explosion is visible from all angles. The polygons are laid out as below:



If you were to draw the three crossed polygons using the default drawing modes the explosion would have a number of 'gaps' in it due to some transparent faces closer to the camera being drawn before faces further away, thus causing the distant polygon to be hidden even though there is no visible part of the close polygon obscuring it. To prevent this it is necessary to override the display mode for the explosion polygons to disable writes to the depth buffer. This causes the polygons to be drawn to the screen without updating the Z values for each pixel, enabling the distant polygons to be drawn 'through' the transparent parts of the closer polygons:





Explosion with Z buffer writing enabled   Explosion with Z buffer writing disabled

In addition to disabling writes to the Z buffer it is also necessary to disable back face culling for the explosion polygons, enabling the image to be seen when looking at the polygons from either direction.

Creating the Geometry for the Explosion

First you should add the following static array of vertex coordinates to `Explosion.cpp` before the `CExplosion()` constructor. These vertices will be used to create the polygon.

```
// Verticies for a 1x1 face in XZ plane, center (0, 0, 0)
static const rtPoint3 sFacePoints[5] =
{
   {-0.5F, 0.0F, 0.5F},
   {0.5F, 0.0F, 0.5F},
   {0.5F, 0.0F, -0.5F},
   {-0.5F, 0.0F, -0.5F}
};
```

Next replace the geometry creation code:

```
    // Create geometry for explosion:
    // Sphere, radius 0.5, full sphere, no capping, 12 segments, using
    // explosionMat
    m_ExplosionGeomID = RTCreateID(RTGeometryID);
    RTSphere(m_ExplosionGeomID, 0.5F, -PiBy2, +PiBy2, RTOff, RTOff, 12,
             m_ExplosionMatID);
```

with the following:

```
    // Create geometry for explosion
    // Create a square polygon:
    rtID polyGeomID = RTCreateID(RTGeometryID);

    // Add the vertex coordinates from sFacePoints to the geometry
    RTPoints(polyGeomID, 4, &sFacePoints);

    rtFace face;

    // Setup values for faces
    face.n = 4;                          // Number of vertices
    face.invis_edges = 0;                // Draw them all when in wireframe mode
    face.material = m_ExplosionMatID;    // Use m_ExplosionMatID for the material

    // Setup the vertex numbers
    face.vertex[0].index = 0;
    face.vertex[1].index = 1;
    face.vertex[2].index = 2;
    face.vertex[3].index = 3;

    // Setup the texture coordinates
    face.vertex[0].texture.x = 0.0F;
```

```
face.vertex[0].texture.y = 0.0F;
face.vertex[1].texture.x = 1.0F;
face.vertex[1].texture.y = 0.0F;
face.vertex[2].texture.x = 1.0F;
face.vertex[2].texture.y = 1.0F;
face.vertex[3].texture.x = 0.0F;
face.vertex[3].texture.y = 1.0F;

// Setup the plane equation and vertex normals for the face
RTSetUpFaces(1, &face, 0, polyGeomID, NULL,
             RTSetUpPlaneEqn | RTSetUpFlat);

// Add the face to the geometry
RTFaces(polyGeomID, 1, &face);
```

Now we have created a polygon with the texture coordinates set so that the image will fill the square. In order to get three way crossed polygons we need to make three instances of the polygon and rotate two of them through 90 degrees. Continuing from the `RTFaces()` call add the following code:

```
// Create three instances of the polygon with 90 degree intersections
rtID polyInstance[3];

for (int i = 0; i < 3; i++)
{
    polyInstance[i] = RTCreateID(RTInstanceID);
    RTAddGeometryAtDetail (polyInstance[i], 0, polyGeomID);
}

// Pitch one placement, roll the other to cross them
rtPosition posn;
RTInqInstance (polyInstance[1], &posn, NULL);
posn.pitch += RMDegToRad(90.0F);
RTSetInstance (polyInstance[1], &posn, NULL);

RTInqInstance (polyInstance[2], &posn, NULL);
posn.roll += RMDegToRad(90.0F);
RTSetInstance (polyInstance[2], &posn, NULL);

// Override back face culling so that the bitmap appears on both sides.
// Also disable Z buffer writing for explosions to prevent the transparent
// bitmap ‘cutting out’ parts of polygons further from the camera
for (i = 0; i < 3; i++)
{
    RTSetDisplayOveride (polyInstance[i], RTBFMask | RTZBWriteMask,
                         RTDisableWriteZB);
}

// Create main explosion geometry - this will be an ‘empty’ shape but will
// have the three instances of the polygon added to it
m_ExplosionGeomID = RTCreateID (RTGeometryID);

// Add the three instances of the polygon to the main geeometry
for (i = 0; i < 3; i++)
    RTAddObject (m_ExplosionGeomID, polyInstance[i]);
```

The source should continue with:

```
// Create an instance to reference the main geometry
m_ExplosionInstID = RTCreateID (RTInstanceID);
. . .
```

In order to prevent the horizontal part of the explosion intersecting with the ground you should find the line that sets the position of the explosion:

```
// Move the explosion to the correct position in the scene
RTSetInstance (m_ExplosionInstID, &pos, NULL);
```

And insert the following before it:

```
// ** Move the explosion off the ground
pos.pos.y += 2.0F;
```

If you now rebuild the application and run it you will notice that sphere used for the explosions has now been changed to intersecting polygons. Now the geometry for the explosion is ready the next step is to add the textured material.

Preparing a Textured Material for the Explosion

Load `Control.rbs` into the STE and create an image using the supplied `TN009_1.TGA` bitmap ensuring that only the ‘Perspective Correct’ and ‘Bilinear Interpolation’ option boxes are enabled. Now create a material called “Explosion Material” - the name is important as we will be using the material from within the API and must use `RTGetIDFromName()` to get the ID. It does not matter what properties this material has as we will be using an image without ‘Mix colour with face’ enabled and this causes the colour of the polygon to be the same as the original bitmap, with no lighting or other calculations applied. Add the image to ‘Explosion Material’ and save the RealiBase - the textured material is now ready to be used on our explosion polygon.

In order to use this new material we must set `m_ExplosionMatID` using `RTGetIDFromName()`. At the top of `CExplosion::CreateExplosion()` replace the material creation code:

```
m_ExplosionMatID = RTCreateID (RTMaterialID);
 RTSetMaterial(m_ExplosionMatID, &sExplosionMaterial);
```

With:

```
m_ExplosionMatID = RTGetIDFromName (“Explosion Material”);
```

If you want to use the explosion multiple times in your application it is necessary to edit `CExplosion::Update()` to replace

```
RTDeleteID (m_ExplosionMatID);
```

With

```
// Reset the explosion material transparency
rtMaterial tempMat;
 RTInqMaterial (m_ExplosionMatID, &tempMat);
  tempMat.transparency = sExplosionMaterial.transparency;
 RTSetMaterial (m_ExplosionMatID, &tempMat);
```

This is necessary as the transparency value is left at 1 (100% transparent) after the explosion has finished so any new explosions would be invisible. Also we do not want to delete the explosion material now as it is no longer created from our program but stored within the RealiBase.

Now the geometry and material for the textured polygon explosions are ready you should build the application, start animating and select ‘Explode Tank 1’ from the menu. The tank should vanish from the view to be replaced by a fireball like explosion.

Adding tank wreckage

At the moment as soon as ‘Explode Tank 1’ is selected from the menu the tank disappears and only the explosion is left in its place. This is not very convincing behaviour for an exploding tank - some wreckage should be left behind. To do this we shall use the STE to add a new object for the wrecked tank to `Control.rbs`.

Load `Control.rbs` into the STE. Create a new view - don’t worry about any of the settings for this view as it is only going to be used as a temporary workspace. In the Views lister expand ‘Main View’ then expand the placement of ‘Tank 1’ below it. Select the ‘T-80 Tank’ geometry and press CTRL-C to copy it. On the options dialog that appears select ‘Copy selected object types from hierarchy’ with ‘Placements and shapes’ enabled. The copy is necessary as you will be editing the geometry and don’t want the original tank to be changed. On the name tag enter ‘Wrecked Tank’ and then check that the AutoView option is set to ‘Do not autoview’. Now when you press OK a new shape called ‘Wrecked Tank’ will appear in your shapes lister. Drag this shape into the temporary view you created and ensure that you can see the tank in that view (select the view window and press F3 for fit-to-view).

Choose Pick | Placements from the menu and then click on the tank's turret - an edit box should surround the turret. Double click on this edit box to enter rotation mode and then drag one of the blue corner markers around so the turret is no longer flush with the top of the tank. Now select the body of the tank and roll that a little way so that it is no longer level - you now have the wrecked tank shape.

Now it is time to give the wrecked tank a burnt-out appearance. Open the materials lister window and scroll through it until you find the 'Black' material. Now open the shapes lister and find the 'Wrecked Tank' shape then expand it. Expand the 'Placement of T80-turret geom 2' and 'Placement of Hull and tracks 2' - You should now see five separate shapes making up different levels-of-detail of the tank and turret. Drag the 'Black' material from the materials lister onto each of the tanks LoD shapes in turn - this will paint all the faces in the shape black.

Now the wrecked tank looks burnt out there's just one more thing to change. In the shapes lister under the 'Placement of T80-turret geom 2' you will see a 'Turret swing' action which you should delete - exploded tanks don't move. Now you have finished creating the geometry for the wrecked tank you can delete the temporary view you created and then save the RealiBase.

Now we need to make a couple of changes to the explosions code to use the wrecked tank model from the RealiBase. Add the following protected member variable to `Explosion.h`:

```
rtID m_WreckageGeomID;      // ID of geometry for wrecked tank
```

Then initialise it in the `CExplosion()` constructor:

```
m_WreckageGeomID = RTNullID;
```

In `CExplosion::CreateExplosion()` after:

```
if (pathid != RTNullID)
{
    // Find current position of an explodingObject on a path
    RTEvaluatePath (pathid, m_StartTime, NULL, NULL, &pos, NULL);
}
else
{
    // Find current position of an explodingObject without a path
    RTInqInstance (explodingObjectID, &pos, NULL);
}
```

Add:

```
// Find the wrecked tank geometry, create an instance, add it to the view
// then move it to the same position as the original tank.
m_WreckageGeomID = RTGetIDFromName("Wrecked Tank");
rtID wreckageInstID = RTCreateID(RTInstanceID);
RTAddGeometryAtDetail (wreckageInstID, 0, m_WreckageGeomID);
RTAddObject (viewID, wreckageInstID);
RTSetInstance (wreckageInstID, &pos, NULL);
```

Now when you run the application and select 'Explode Tank 1' from the menu you will see that the burnt out wreckage replaces the original tank as the explosion starts - the explosion is now complete.

Further Improvements

The application presented in this document could be enhanced in the following ways:

- use animated textures for the explosion polygons. Using a sequence of images of a progressing explosion you could change the material used by the polygon to one with a different bitmap each frame producing a more active explosion. An alternative method of achieving the same effect is to use one image containing several frames of the explosion

and alter the texture coordinates for the polygon each frame so that only the part of the bitmap with the desired image can be seen.
- Allow multiple simultaneous explosions by maintaining a list of geometries and materials for the separate explosions.
- Attach a sound to the explosion
- Animate the target as it explodes - e.g. turret lifting off and crashing to earth
- Use different explosion classes by turning `CExplosion` into a base class and deriving other explosion types from it.

Summary

In this document we have shown how an object can be made to disappear from the view and be replaced by alternate geometry. Geometry creation has been demonstrated using standard graphics primitives like the sphere and by creating individual polygons from specified vertices. This geometry has then been animated to create an explosion effect. For further examples of special effects within RealiMation see the RealiGame source code sample which includes functions to generate trails behind objects suitable for vapour trails from aircraft wingtips or missile smoke trails.