

# Roids – Game Design and Implementation

Twm Davies

Published by the Symbian Developer Network

Version: 1.0 – August 2007

<b>1 BACKGROUND</b> .....	<b>2</b>
1.1 DISPLAY AND TERMINOLOGY .....	2
1.2 DESIGN PHILOSOPHY .....	3
1.3 BRIEF DESCRIPTION OF THE GAME'S CLASSES .....	3
<b>2 THE GAME LOOP</b> .....	<b>4</b>
2.1 OVERVIEW .....	4
2.2 KEY PRESS HANDLING .....	7
2.3 SHAPES AND DRAWABLE SHAPES .....	8
<b>3 DRAWING THE GAME</b> .....	<b>11</b>
3.1 OFF-SCREEN BITMAP .....	11
3.2 DRAWING EXPLOSION FRAGMENTS .....	12
3.3 DRAWING SHIP EXPLOSION (AKA THE STARBURST) .....	13
<b>4 PERFORMANCE AND MEMORY CONSIDERATIONS</b> .....	<b>13</b>
4.1 MEMORY ALLOCATION .....	13
4.2 ROIDS MATHS .....	15
<b>5 PORTABILITY</b> .....	<b>17</b>
5.1 GENERAL ISSUES .....	17
5.2 SUPPORTING SERIES 80/S60 AND UIQ IN CARBIDE.C++ .....	18
<b>6 OTHER OBSERVATIONS</b> .....	<b>20</b>
6.1 DEBUGGING AIDS .....	20
6.2 SVG ICON DESIGN .....	21
<b>APPENDIX A - CODE TO PRE-GENERATE SIN AND COS</b> .....	<b>22</b>
<b>APPENDIX B - EXERCISES</b> .....	<b>23</b>
<b>ABOUT THE AUTHOR</b> .....	<b>23</b>

# 1 Background

Roids is a 2D game clearly 'inspired' by the classic vector-based arcade game Asteroids, released in 1979 by Atari. The player controls a triangular ship which is trying to pick up capsules in the middle of an asteroid storm. The game is made up of a series of levels which are cleared by shooting all the roids and picking up each capsule. However when a roid is hit, it splits into two smaller roids causing more objects to avoid which is not so good because one touch and your ship is destroyed. Roids has nice, over-the-top explosions and introduces tougher rocks, which appear in later levels and take two or more shots to split.

The game was originally written in one day (downing tools at 6am) as a challenge after witnessing a colleague take weeks and weeks to write an over-designed object-oriented asteroids clone. The goal was to write a slick playable game on the Nokia 9210 using the Symbian application framework and drawing code. I also wanted to try writing an object oriented game.

The purpose of this document is to explain some of the design and Symbian-based implementation details, including optimisations. The target audience is those who have never written a game before or those first starting out on Symbian OS. The source can be used as a teaching example and exercises are provided in the appendix for further expansion on the game.

## 1.1 Display and terminology

Figure 1 shows a screen shot from the Roids game and highlights the main elements involved in screen.

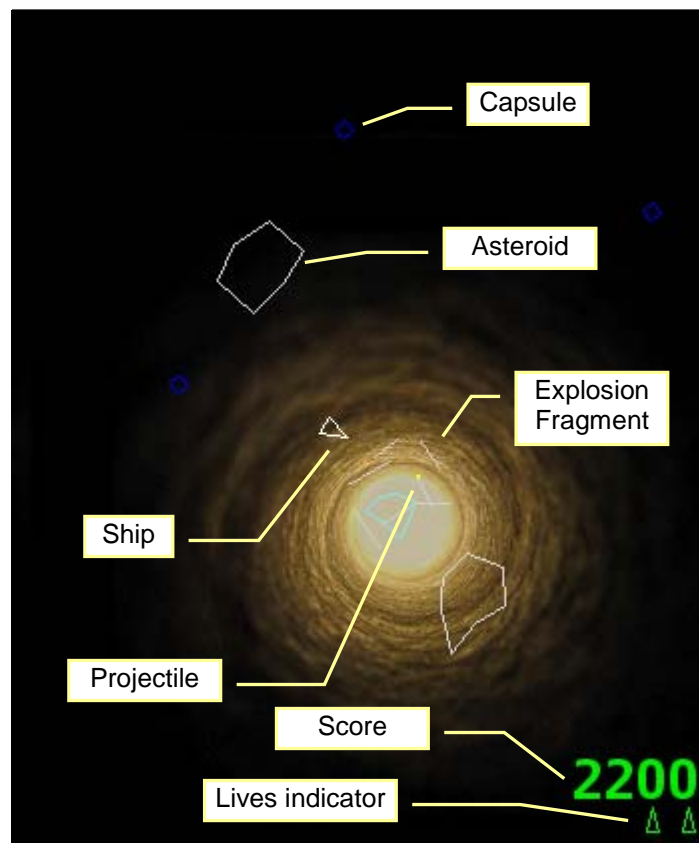


Figure 1 The Roids game main display

## 1.2 Design philosophy

Asteroids is not a particularly original game to copy, but it makes a great teaching example, showing how to implement a slick game on a mobile platform. One of the great features of asteroids is that levels can be derived algorithmically. This means that the levels don't have to be laboriously designed and tested. We can simply say *number of asteroids = level number* and get on with tweaking the game play and performance.

The game has to perform well and offer very responsive control. The intent was to achieve the performance while also making sensible use of Symbian OS classes and employing sensible Object Oriented programming techniques. The game is implemented as a good citizen on a device; it does not dominate the CPU and has an active object-based game loop which allows standard menus and other UI services to be used.

## 1.3 Brief description of the game's classes

### 1.3.1 Simplified class description

Roids is a standard Symbian application with one full screen control, `CRoidsContainer`. The `CRoidsContainer` is the central class which owns all objects and is responsible for drawing and handling input. In MVC (model-view-controller) terms, `CRoidsContainer` does all three; it owns the representation of the world, it is responsible for drawing it and it responds to input events.

Figure 2 shows how the classes fit together. There are many `TWorldObject`-derived objects in the Roids game, but only `TShip` is shown for clarity.

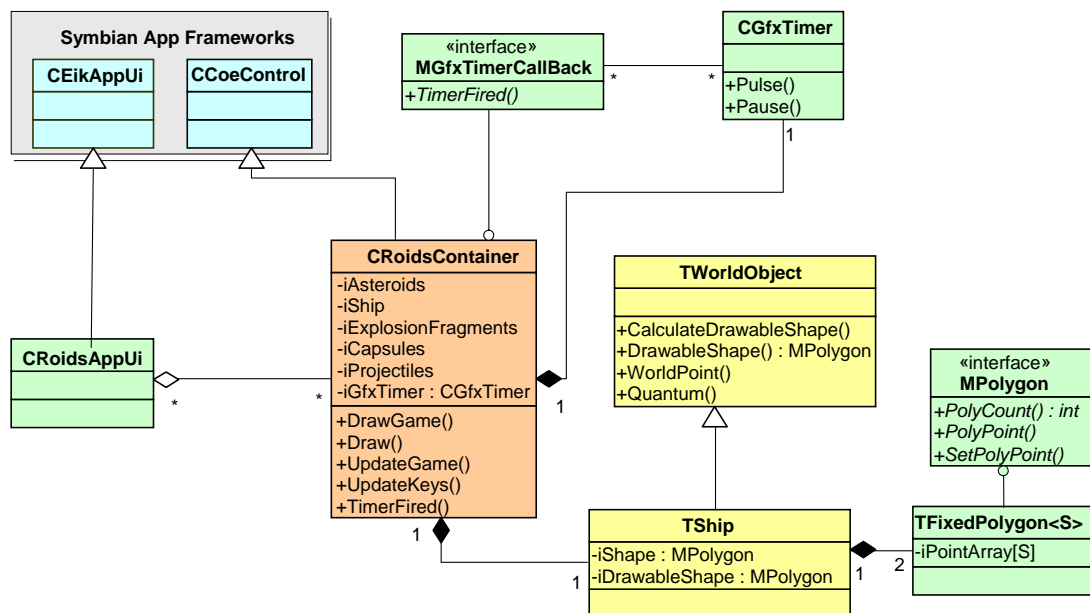


Figure 2 Simplified UML class diagram

To quickly summarise: when the timer fires, the container updates the world objects, reads the state of the keys, recalculates the shape and detects collisions before drawing and starting again.

### 1.3.2 World objects

World objects are polygon-based shapes which move around in the world; these are asteroids, projectiles, capsules and the ship. `TWorldObject` is the base class for all objects and provides methods for placing and manipulating the object. An object has a polygon representation of the shape and has the speed and direction of travel of the object. World objects are fairly abstract; they do not know how to draw themselves and have no idea about boundaries of the world or about collision.

The `CRoidsContainerClass` is responsible for checking collisions and makes use of `TWorldObject::Intersects(TWorldObject aObject)`. The container also wraps objects around when they approach the side of the screen and is responsible for drawing them.

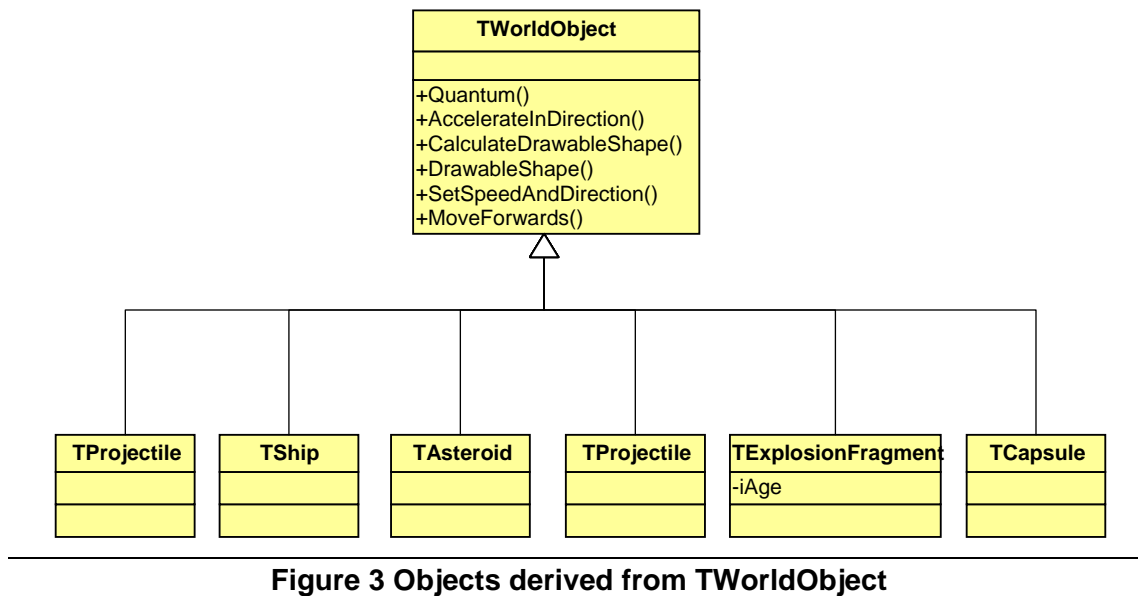


Figure 3 Objects derived from `TWorldObject`

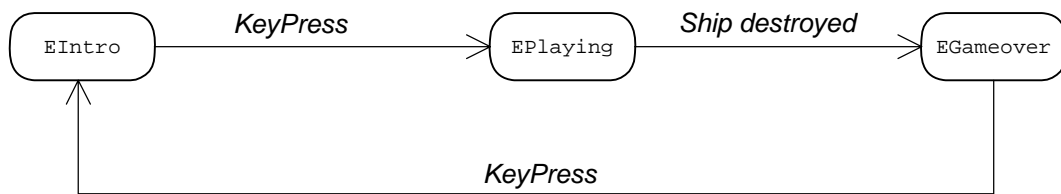
## 2 The game loop

### 2.1 Overview

#### 2.1.1 Game states

The game has 3 high level states

1. `EIntro` – displays the splash screen with Roids moving around the background
2. `EPlaying` – the main game loop
3. `EGameOver` – Roids continue moving around until a key is pressed but the ship has been destroyed




---

**Figure 4 The game states**

---

### 2.1.2 The main game loop

The main game loop is perpetual and runs in all three high level states. It consists of seven logical steps which are performed on each cycle of the game loop. Here is the pseudo code for the loop:

```

FOREVER
    WaitForHeartBeat()
    UpdateFromKeyPress()
    Quantum()
    CalculateDrawableShapes()
    CalculateCollisions()
    DrawShapes(backupbitmap)
    RedrawControlFromBackupBitmap()
END
  
```

The rest of this paper goes through each logical step of the loop, starting with the heartbeat.

### 2.1.3 Timers and ticks

There are two ways of implementing time-based events in a game:

1. Use OS timers for each event.
2. Use ticks based on the number of times the game state has been updated.

An example of a timer being used would be to time a minute of play in order to penalise the player for loitering. When using OS timers for such events, there may be many events firing which lead to changes to the state of the game world.

#### 2.1.3.1 Problem

Using OS timers for all time-based events can lead to very complex code, which is difficult to debug and puts a strain on resources, since hundreds of objects may need some sort of timing-related service.

Another problem is that of accuracy; the time to draw something may vary and so times may begin to drift. It's also not easy to synchronise events to a time code – for example fading a graphic over time.

#### 2.1.3.2 Solution

A heartbeat timer is used to synchronise the game. This timer fires at a rate which approximates the desired frame rate, say 30 frames per second, giving a 30 times a second heartbeat.

The heartbeat itself is a normal OS timer, which keeps a 'tick count' of the number of times it has fired. All other elements of the game which need placement in time specify time as a tick count.

The whole game can be paused instantly or played in slow motion simply by reducing the speed of the heartbeat (which is very useful for debugging collisions). This also makes the game perform at the same speed on all devices.

In order to achieve a fading effect as an object explodes, when a rock fragment effect is created, the fragment has its age member set to zero. On each heartbeat, the age is incremented and is used to calculate a varying grey scale when drawing the shape with 0 = white and 255 = black. When the age hits 255, the fragment is removed from the world. By assigning an age to each fragment, the game can deal with hundreds of fragments from a single OS timer.

The heartbeat timer is pretty standard. `CGfxTimer` is a very simple implementation which wraps `CTimer` with call through to a Mixin class `MGfxTimerCallback`.

`StartPulse()` causes the timer to fire continuously at a specified interval.

```
class MGfxTimerCallback{
public:
    virtual void GfxTimerFiredL(TInt aId) = 0;
};
class CGfxTimer : public CTimer
{
public:
    CGfxTimer(MGfxTimerCallback& aCallbackClient, TInt aId);
    void ConstructL();
    void StartPulse(TTimeIntervalMicroSeconds32 aInterval);
    void Start(TTimeIntervalMicroSeconds32 aInterval);
    void Pause();
    void Restart();
    virtual void RunL();
}
```

The EKA1 emulator can only service timers at a granularity of a 1/10 of a second, whereas on-target hardware timers are serviced at 1/64 of a second. The effect of this is a particularly slow game in the emulator - it runs at just 10 frames per second. The timer granularity in the Emulator was improved in EKA2 and much more accurately simulates the target platform.

### 2.1.4 The quantum

The quantum is what makes the world move. Each object in the world has a virtual `Quantum()` function. This function is called for each frame and tells the object to update its state given that one time slice has passed. It ensures that Roids rotate smoothly with each frame and the ship continues to move after the thrust key has been depressed. `Quantum()` is called on each object before calculating collisions.

It's completely up to the object what it does with a quantum. If the function is left empty, then the object simply remains stationary. Here's how various objects handle their quantum:

Object	Quantum
TAsteroid	Move the centre point in the direction of travel. Increment the angle by 5
TExpl osionFragment	Move centre point by the current speed in the direction of travel
TProject ile	
TShi p	
TCapsul e	Rotate, and pulse the scalar

## 2.2 Key press handling

### 2.2.1.1 Problem

The Symbian Window server (WSERV) is responsible for delivering key events to the application. WSERV uses a typical GUI paradigm designed for controls such as text editors and list boxes. Holding a key down is reported in terms of an initial event followed by a delay and then a number of repeat key events.

This does not work for games, where responsive and fluid control is needed. A game environment may also have to deal with two or more keys pressed simultaneously, as is the case when directing the ship to the right and forward at the same time.

### 2.2.1.2 Solution

As well as key and key repeat events, WSERV reports key up and key down events. These can be used to toggle a flag indicating that a particular key is being held. By remembering the state of relevant keys, the key repeat information can be ignored.

At each heartbeat, the state of the key is checked and any rotation or acceleration is applied. For ship left and right keys, the facing angle of the ship is incremented or decremented on each heartbeat that occurs while a left or right key is pressed. This results in smooth control of the angle of the ship.

When the player accelerates, the ship is accelerated in the direction it's facing by a constant amount each heartbeat. This allows the user to turn the ship around and accelerate in the opposite direction of travel in order to slow down the ship.

Each game key, which we'll call *virtual keys*, is defined in an enum:

```
enum TGameKey
{
    EKeyRotateLeft,
    EKeyRotateRight,
    EKeyThrust,
    EKeyFireGun,
    EKeyMaxVirtualKey
};
```

```
TInt iScanCodes[EKeyMaxVirtualKey];
```

`iScanCodes` is an array which records the status of a virtual key. A zero in the array indicates that the virtual key is not currently pressed, while a non-zero value represents the scan code of the actual key associated with the virtual key. This scheme allows more than one physical key to trigger a virtual key and supports multiple keys pressed simultaneously.

```
void SetKeyDown(const TGameKey& aKey, const TKeyEvent& aKeyEvent);
void ClearKeyDown(const TGameKey& aKey);
TBool KeyDown(const TGameKey& aKey);
TKeyResponse OfferKeyEventL(const TKeyEvent& aKeyEvent, TEventCode aType);
```

The key handling is now very simple. Three functions `set`, `clear` and `return` the status of the virtual key. The overridden `CCoeControl::OfferKeyEventL()` simply maps key up and key down events to calls to `SetKeyDown()` and `ClearKeyDown()`.

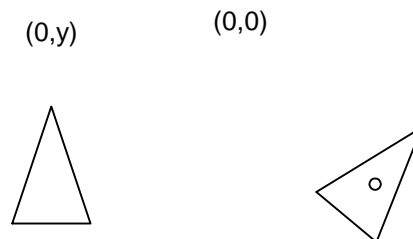
The actual key handling is decoupled from the UI event into the `UpdateFromKeyStates()` method. The method is called on each heartbeat (rather than each key event) and checks each virtual key using `KeyDown()`, and then applies any changes to the game world before it is drawn. Typical tasks of this method are to create a new projectile object in response to a `EKeyFireGun` event, or

to increment and decrement the ship's angle in response to a EKeyRotateLeft or EKeyRotateRight event.

## 2.3 Shapes and drawable shapes

A world object has two representations; Each object has a 'master' definition shape which describes the polygon shape in terms of an array of vertexes around a centre point of (0,0).

Whenever the object needs to be drawn on screen the master shape is copied, rotated and moved to calculate the position and shape of the object in the real world.



**Figure 5 Shape vs drawable shape**

As a performance optimisation, the drawable shape is cached as part of the object since it's used for both collision detection and drawing. The cached drawable shape must be recalculated each time the object's angle or position changes.

Drawable shapes are called so because they can be drawn to the world without any further transformation. The MPolygon class contains all the functions needed to transform a shape into a drawable shape. All the methods are const since all changes are applied to a destination polygon.

```

Class MPolygon:
...
void Rotate(TInt aAngle, MPolygon &aDestination) const;
void GetBoundingRect(TRect& aRect) const;
void RotateAndOffset(TInt aAngle, TPoint aOffset, MPolygon& aDestination)
const;
void RotateOffsetAndScale(TInt aAngle, TPoint aOffset, TReal iXScale, TReal
iYScale, MPolygon& aDestination) const;
void Offset(TPoint aOffset, MPolygon& aDestination) const;
...

```

### 2.3.1 Generating asteroids

Each asteroid is uniquely generated with a simple randomiser algorithm. The basic algorithm is similar to drawing a circle.

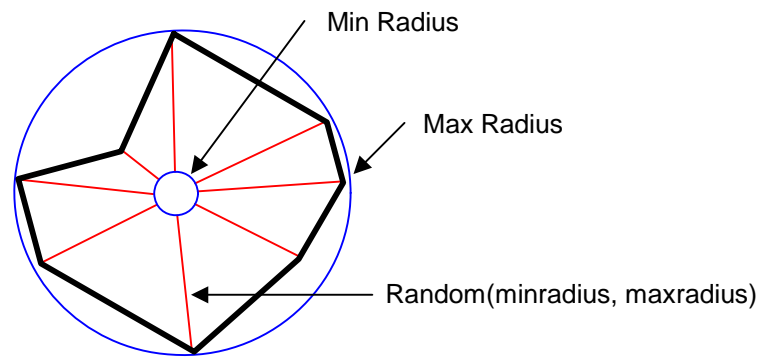
To plot the circumference of a circle of radius 30 we might have the following pseudo code

```

Radius = 30
For angle = 0 to 360
    x = radius * -sin(angle)
    y = radius * cos(angle)
    plot(x, y)
EndFor

```

The Roids asteroid generator first divides the circle into a random number of segments; there are eight in the example shown in Figure 6.




---

**Figure 6 Calculating asteroids shapes**

---

For each segment a random radius is chosen between a minimum and maximum range and a value for x,y is found. The x,y values form vertices for the polygon.

The point of the minimum and maximum is to assert some control over the general size of the asteroids. When an asteroid splits, we may want to create two asteroids of roughly half the size of the original asteroid so the radius of the large asteroids is divided by two.

In terms of look, the difference between minimum and maximum defines how spiky the asteroid is.

### 2.3.2 Shooting projectiles

When the player fires the gun, a projectile must appear to shoot from the nose of the ship in the direction it's pointing.

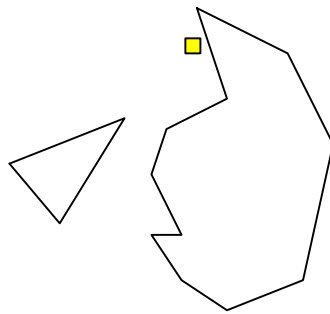
When a projectile is created, the direction of the projectile is copied from the ship and a helper function in TShip returns the coordinates of the nose. The speed of the projectile is a constant regardless of how fast the ship is travelling.

```
void CRoidsContainer::LaunchProjectile()
{
    TProjectile* proj = iProjectilePool->NewObject();
    if(!proj)
        return; // Ran out of projectiles
    TInt direction = iShip.FacingDirection();
    proj->SetActive(ETrue);
    proj->SetWorldPoint(iShip.Nose());
    proj->SetSpeedAndDirection(direction, 10);
}
```

### 2.3.3 Collision detection

#### 2.3.3.1 Problem

Mathematically, detecting collisions between static polygons is pretty straightforward. However in a game environment there is the added complication that an object may be travelling at several pixels per second making it possible for high speed objects to travel through each other.




---

**Figure 7 Problems with collision detection at high speed**

---

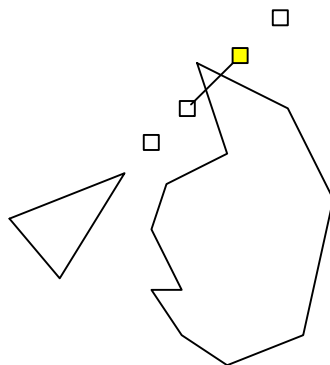
The problem is illustrated by Figure 7. The grey boxes show how the position of the projectile changes rapidly between frames. According to the laws of physics, there should be a collision between position 2 and 3, but it will not occur if a simple polygon intersection test is made at each point.

This may seem trivial attention to detail, but it has a detrimental effect on the playability and has to be considered.

### 2.3.3.2 Solution

More reliable time based collision detection can be achieved by tracing the route of the projectile to see if a collision should have taken place between the current frame and the last.

On each frame a line is constructed for each projectile from the current point of the projectile and the previous point. The whole line is tested for collision with the asteroid as shown in Figure 8.




---

**Figure 8 Reliable collision detection using a line between the current and previous points**

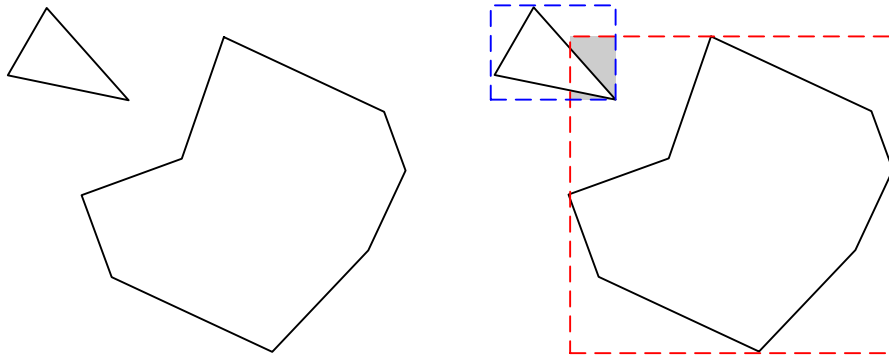
---

### 2.3.4 Optimising collision detection

Detecting collisions between two polygons is a relatively expensive operation. It involves multiple loops around vertices and lines in order to work out if the two shapes overlap. If the ship-to-asteroid detection and projectile-to-asteroid collision was done for each frame, then it could dramatically slow down the frame rate.

Most of the time consuming calculations are redundant, since the asteroids are most likely to be minding their own business at some distance from the ship or projectile, and so the first optimisation is to construct a bounding rectangle around the ship and each asteroid.

The test for intersection of two rectangles is trivial compared to a full polygon intersection and quickly eliminates rocks which are nowhere near any other object.



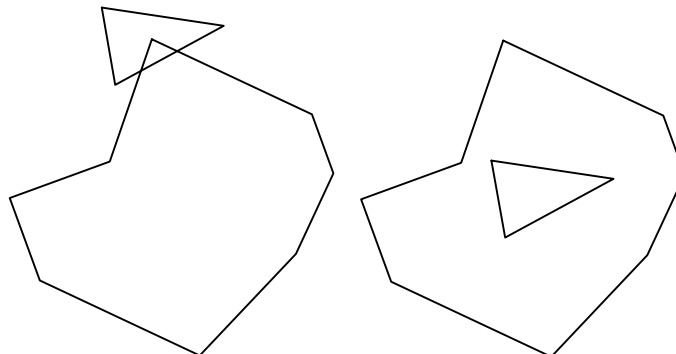

---

**Figure 9 Quick test for possible candidate for collision**

---

Reliably detecting a hit between two polygons is quite tricky. At first it seems that testing for line intersections between all lines in both polygons would work, but this fails if one polygon is fully within the other.

To test reliably we have to test each vertex in the polygon A for inclusion in polygon B and vice versa. This takes into account all possibilities.




---

**Figure 10 Line intersection**

---

### 3 Drawing the game

Once all the objects have been recalculated and collisions detected, all that's left before starting again is to draw the display so that the user can see the new state of the world.

Since Roids is based on wire frame vectors, the Z order for drawing objects is not important, for example, when two asteroids are overlapping, it makes no difference in which order they are drawn – the same pixels are drawn.

#### 3.1 Off-screen bitmap

The `CRoidsContainer` object owns a scratch pad bitmap exactly the size of the control. All drawing is built up on this bitmap until the last object is drawn and the bitmap is copied to the screen. The off-screen bitmap provides 'atomic' update of the screen which eliminates flickering.

The pseudo code sequence for drawing the game is.

```

DrawGame(bitmap)
DrawBackgroundImage()
DrawStarBust()
DrawFragments()
DrawProjectiles()
DrawAsteroids()
DrawLivesAndScoreIndicator()
DrawShip()
DrawCapsules()
DrawNow() // ← forces the window server to request Draw()
END

```

The draw() function is incredibly simple

```

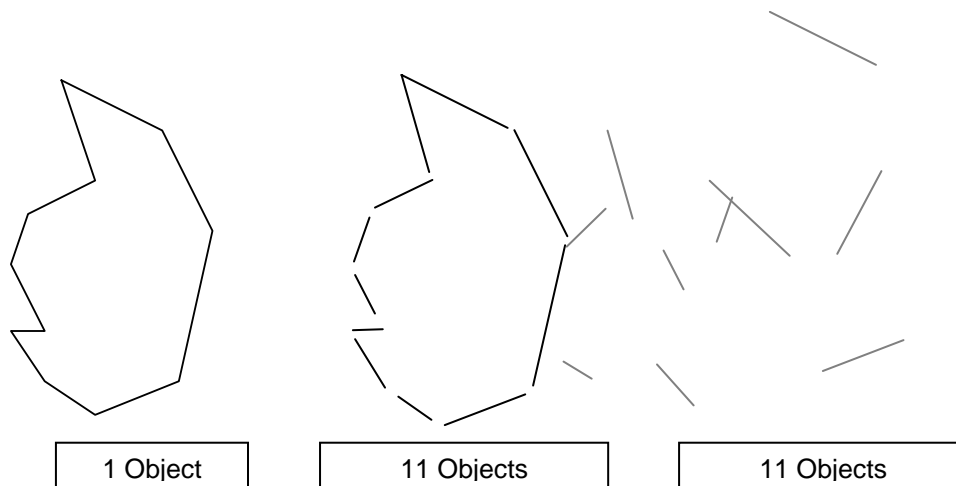
void CRoidsContainer::Draw(const TRect& aRect) const
{
    CWindowGc& gc=SystemGc();
    gc.BitBlt(aRect.iTL, iBackupBitmap, aRect);
}

```

### 3.2 Drawing explosion fragments

When a projectile collides with an asteroid, it causes an explosion effect whereby fragments of the asteroids are propelled from the crash site and gradually fade away into space. Fragments are passive and don't interact or collide with other objects in the world, they are purely for show.

The effect is achieved by tracing the shape of the asteroid and creating a TExplosionFragment object for each line. Each fragment is given an age which is incremented on each heartbeat which allows the colour of the fragment to be faded over time.



**Figure 11 Making explosions**

The algorithm takes any TWorldObject and creates TExplosionFragments for it. Each fragment is given the speed direction of its parent object so that it explodes in the direction of travel. In addition, we also apply a random factor to the direction in order to scatter the fragments.

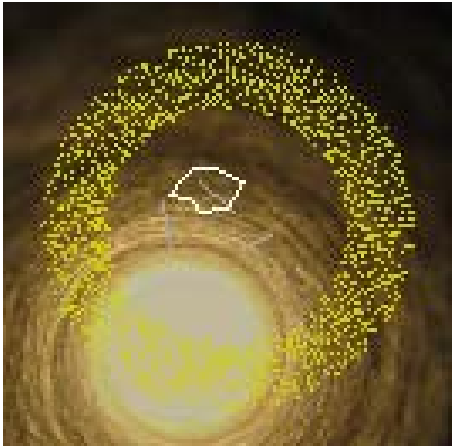
```

void CRoidsContainer::CreateExplosionFragmentsFromObject(TWorldObject& aObject)

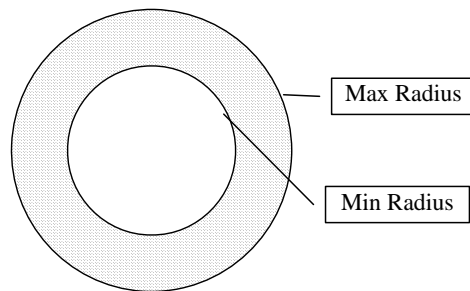
```

### 3.3 Drawing ship explosion (aka the starburst)

When a ship collides with Roid, it exits in burst of burning fuel. This effect is handled differently to fragment explosion.



**Figure 12 Starburst**



**Figure 13 The general shape of the starburst**

The still from the game in Figure 12 shows that the explosion is made up of two concentric circles filled with yellow pixels between them.

The centre point of the starburst is taken from the time a ship was destroyed. The point continues to move in the same direction as the ship, 'picking' up the energy stored in the ship as it accelerates. This effect is so cool you might want to crash the ship just to see it!

Yellow pixels are selected by walking around the circle and for each angle picking a random point between the minimum and maximum radius. This is repeated until a dense cloud of particles is visible. The advantage of being randomly generated is that the starburst seems to shimmer as it expands.

The actual pixels in the explosion are plotted using the Symbian class `TBitmapUtil`. This bypasses the usual graphics context in order to plug pixel values directly into the bitmap memory. This is far more efficient than trying to plot using a 1-pixel-wide brush etc.

## 4 Performance and memory considerations

### 4.1 Memory allocation

In Roids, it would make no sense for the game to run out of memory when trying to shoot a projectile or split an asteroid. Memory may become exhausted at any time on a multitasking machine.

Since the memory usage of most games can be predicted, it's far better to fail at application launch time and ask the user to shut down other applications or, even better, let the device's memory manager sort it out for you<sup>1</sup>

<sup>1</sup> Symbian OS-based phones have a low memory watchdog which shuts down applications in the background in order to free resources to launch new applications.

### 4.1.1 Allocate at Start-up

The main control in Roids allocates all the space it needs for world objects such as asteroids and projectiles. If all the memory required cannot be allocated then a `Leave(KErrNoMemory)` will be generated which stops the application from launching.

Roids allocates all memory in the `CRoidsContainer::ConstructL()` and all further allocations are done from memory pools.

### 4.1.2 Memory pools

Many objects come in and out of existence during the lifetime of a game. When a roid is hit, the object is freed and two smaller roids come into existence - increasing the amount of memory required to represent the roids. If dynamic allocation was used then there is always the chance that allocation will fail.

Memory pools are just pre-reserved blocks of memory which do not grow or shrink and are assumed to be large enough to hold all the memory required during the game's lifetime.

A quick way of retrofitting a memory pool to an application is to set the minimum size of the heap to cover the memory requirements of the application. If the process succeeded in launching then there is clearly enough RAM reserved in the heap for all memory allocations. The advantage is that you can avoid changes to code; the disadvantage is that you have to profile every code path or just take an educated guess.

Roids makes use of explicit memory pools which are efficient fixed sized arrays for each type of world object. A second array keeps track of whether an array item is valid or not.

The allocation table for a pool of up to four objects with two current allocations looks like this:

Object	Valid
<code>TAsteroid[0]</code>	True
<code>TAsteroid[1]</code>	True
<code>TAsteroid[2]</code>	False
<code>TAsteroid[3]</code>	False

The pool implemented as a generic type `CObjectPool` which can store a pool of any type of object of any fixed size.

```
template<class T, TInt S>
class CObjectPool : public CBase
{
public:
    CObjectPool () {iSize= S;};

    T* NewObject();
    void FreeObject(T* aAsteroid);

    void Reset();
    TInt Count() const;
    T* Object(TInt aIndex) const;
    TPoolIterator<T, S> Iterator();

private:
    friend TPoolIterator<T, S>;
    T iObjects[S];
};
```

```

    TUint8    i FreeMap[S];
    TInt     i Size;
    TInt     i Count;
};

```

In order to declare a pool of up to 30 asteroids, the following is used.

```
CObjectPool<TAsteroid, 30>* asteroidPool = new(ELeave) CObjectPool<TAsteroid, 30>;
```

An example of the usage is:

```

TAsteroid* asteroid = asteroidPool ->NewObject();
// do something with the asteroid
asteroidPool ->FreeObject(asteroid);

```

The pool is also an array-like container, and an efficient iterator is provided to perform operations on all allocated objects.

```

template<class T, TInt S>
class TPoolIterator
{
public:
    TPoolIterator(CObjectPool<T, S>* aPool):
        T* First()
        T* Next();
};

```

The sizes of the pools are defined as convenient typedefs to save continuously inserting the size of the pool into code.

```

typedef CObjectPool<TAsteroid, 30> CAsteroidPool;
typedef TPoolIterator<TAsteroid, 30> TAsteroidIterator;
typedef CObjectPool<TCapsule, 20> CCapsulePool;
typedef TPoolIterator<TCapsule, 20> TCapsuleIterator;
typedef CObjectPool<TProjectile, 20> CProjectilePool;
typedef TPoolIterator<TProjectile, 20> TProjectileIterator;

```

If a pool fails to allocate an object then it ASSERTs, alerting the developer that not enough space has been set aside.

## 4.2 Roids maths

Physics-heavy games make use of floating point numbers to model acceleration, rotation and scaling. Current ARM architectures found on mobile phones are not so great at floating point calculations since FPU instructions are not natively supported in hardware and have to be emulated in software. This means that floating point calculations for operations which need thousands of calculations per seconds can be quite time consuming.

Rarely is great precision required for expensive geometric operations such as sin(), cos() and square root; in practice 1 or 2 decimal places would suffice, so there is room for optimisation for limited game use.

There are several optimisations for sin and cos; for example, the Taylor series can be used to quickly approximate sin and cos to a number of decimal places.

$$\sin(x) = 0 + 1 \times x - 0 \times \frac{x^2}{2!} - 1 \times \frac{x^3}{3!} + \dots$$

---

**Figure 14 Taylor series for rapidly approximating sin**

---

Roids specifies angles in degrees and limits rotation to whole angles from 0-360. Such a discrete range can be stored in a lookup table for rapid retrieval.

### 4.2.1 MathsUtil

The MathsUtil class is a static utility class which wraps the look-up data and the functions.

Before any functions are used, a user of the class must call ConstructL(), which allocates memory to pre-generate the lookup tables, and Reset() to free the memory:

```
MathsUtil::ConstructL()
MathsUtil::Reset()
```

### 4.2.2 Maths functions provided by MathsUtil

Function	Description	Example Use
TReal Sin(TReal aRadian)	A simple wrapper over the standard E32maths. This does not use a lookup table	Generating random asteroid shapes
TReal Cos(TReal aRadian)	A simple wrapper over the standard E32maths. This does not use a lookup table	Generating random asteroid shapes
TReal Sin(TInt aAngle)	Return the sine of a discrete angle between 0 and 360.	Calculating the rotation of the ship
TReal Cos(TInt aAngle)	Return the Cosine of a discrete angle between 0 and 360	Calculating the rotation of the ship
TInt SquareRoot(TInt aNum)	Fast integer square root of aNum	Calculate the distance between objects
TInt Distance(TPoint aA, TPoint aB)	Returns the distance between two points	Not currently used, but intended to perhaps play warning sound when ship is in proximity of a roid.
TBool Intersect(a, b, c, d)	Returns true if line a->b intersects with line c->d	Polygon intersection code to detect collisions.
TInt Round(TReal aNumber)	Fast rounding from real to TInt	All real to int conversions.
template<class T>inline T Bound(T x, T a, T b)	Return X bounded by the minimum value a, and the maximum value b	
Random(TInt aA, TInt aB)	Return a random number in the range aA to aB	Generating asteroids shapes limiting the shape between a minimum and maximum radius

### 4.2.3 Static lookup data

At the cost of a slightly larger binary size, the lookup data can be pre-generated on a PC and stuffed into a header file. The data does not have to be writable and so works on all versions of Symbian OS.

(Appendix A - Code to pre-generate sin and cos shows the simple program used to generate the data.)

## 5 Portability

Roids is remarkably easy to port to different devices and UIs. The game automatically adjusts to different screen sizes and requires no platform security capabilities in order to run.

The original game was written for the 9210, and the original SIS file still installs happily on the latest communicator models (the 9500 and 9300). It has also been successfully compiled for the 7650 and most recently the S60 3<sup>rd</sup> edition emulator.

### 5.1 General issues

#### 5.1.1 Platform Security

Roids does not need any special privileges and simply draws to a standard `CoeControl`. It does not make use of direct screen access and does not adjust the thread priority.

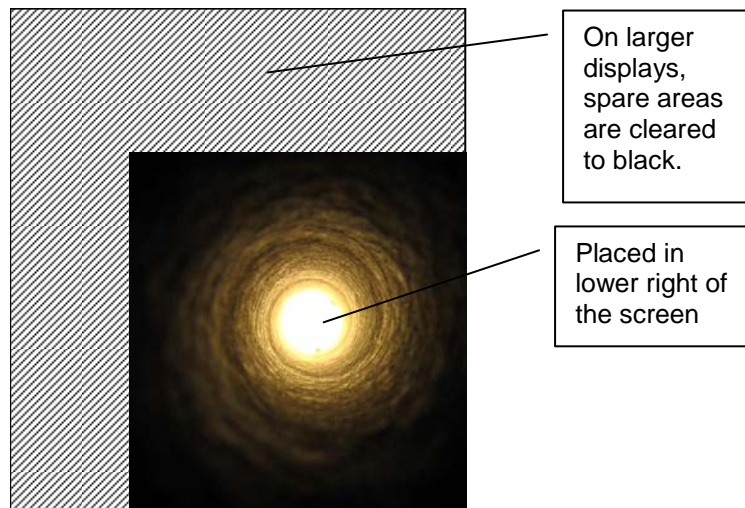
#### 5.1.2 History

The first implementation of MathsUtil s required thread local storage (TLS) since it pre-generated sin and cos tables at application start-up. This has been eliminated by using a pre-generated header file with the lookup tables. A small penalty of a few kilobytes of lookup data built into the binary is a valid trade off against conditional code for EKA1 and EKA2 platforms.

#### 5.1.3 UI scaling

The game world is always a full screen `CCoeControl`. There are no assumptions about the size of the control in the game; all the logic for placement and wrapping around of world objects retrieve the width and height of the control.

If the bitmap was scaled to full screen then it will distort for screen sizes with different aspect ratio. So the background bitmap is oriented to the bottom right corner of the screen and any gaps will be filled in black.



**Figure 15 Bitmap construction for different sized displays**

#### 5.1.4 Key handling

The game uses virtual keys internally and relies on the control mapping window server keys to the virtual keys. Since handsets come in different shapes and sizes, it's hard to come up with a

mapping which works for all devices. In particular the fire button is hard to choose since we don't want it too close to the joystick. In order to get around this problem, the fire key is mapped to any key aside from the cursor/joystick.

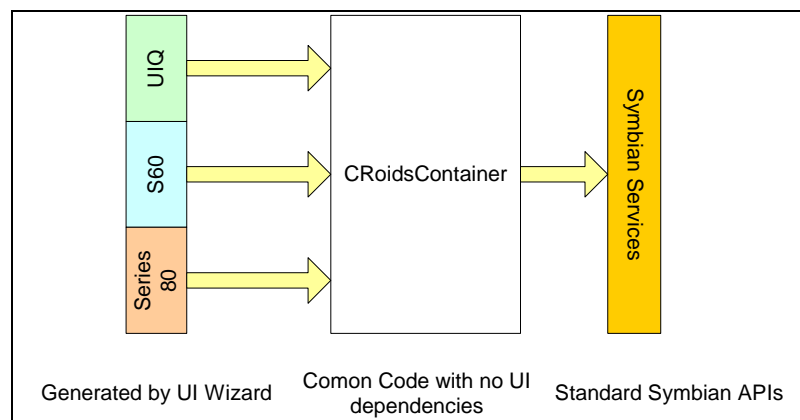
Virtual Key	Mapping
EKeyRotateLeft	Left cursor/joystick
EKeyRotateRight	Right cursor/joystick
EKeyThrust	Up cursor/joystick
EKeyFireGun	Any other key

## 5.2 Supporting Series 80/S60 and UIQ in Carbide.c++

The original strategy for Roids portability was to derive from Symbian application base classes such as `CEikAppUI` and `CEikDocument`. However UIQ and S60 have diverged significantly enough as of version 3 of the platforms so as to require specific base classes. In addition, the build tools have diverged for supporting application icons etc., which makes it increasingly hard to support several platforms from a single code base.

While developing using Carbide.c++, it became clear that the differences were irreconcilable.

Carbide.c++ introduces new application wizards for creating the boiler-plate and repetitive code for each platform. The wizards provide a much more reliable way of getting it right for a target platform



**Figure 16 Supporting different UI platforms**

Figure 16 shows the partitioning. The majority of the Roids code is portable, with dependencies on Symbian APIs and services which are available on all platforms.

The Roids API is exposed as a `CCoeControl`, and each UI variant must simply add the control, setting the rectangle to the dimensions which make sense on the device.

For Series 80 and S60 the `ApplicationUI` adds the `CRoidsContainer` as the sole control and sets it to full screen. The UIQ version on the other hand has a `View` class (created by the wizard) which aggregates the `CRoidsContainer` and translates key events from the rocker into cursor events before handing them to the `CRoidsContainer`.

Although there is a cost to maintaining several UI variants, the time saved in porting makes it worth while and allows each platform to be tweaked for the right look and feel for that UI. Roids only has one menu item and so the code churn in the application-specific code is very low.

To keep a design clean and portable, the API UI class should not do any significant processing, but just do command forwarding and mutation of key events according to the specific hardware. When supporting multiple platforms, consider creating a DLL which holds the engine code. This makes it very clear where the line of portability should be drawn.

### 5.2.1 Porting S60 to UIQ

The UIQ port was realised very rapidly by creating a blank project called RoidsUIQ. The .cpp files and .h files were dragged into the work space and built to ensure everything compiled in the new workspace.

The View class created by the wizard was then modified to add the Roids container.

```
void CRoidsUIQView::ViewConstructL()
{
    // Loads information about the UI configurations this view supports
    // together with definition of each view.
    ViewConstructFromResourceL(R_UI_CONFIGURATIONS);

    CRoidsContainer* container = new(ELeave) CRoidsContainer();
    AddControlLC(container); // takes ownership
    container->ConstructL(iQikAppUi.ApplicationRect());
    CleanupStack::Pop(container);
    iRoidsContainer = container; // does not take ownership
}
```

The resource file was modified to stop any additional controls being created.

```
RESOURCE QIK_VIEW r_RoidsUIQ_Layout
{
    //pages = r_RoidsUIQ_Layout_pages;
}
```

### 5.2.2 Mapping key events

UIQ uses special key codes for the joypad-like control, which need to be mapped to cursor keys before they are passed to the roids control.

```
TKeyResponse CRoidsUIQView::OfferKeyEventL(const TKeyEvent& aKeyEvent, TEventCode aType)
{
    TKeyEvent event(aKeyEvent);

    TInt newCode = event.iScanCode;
    switch(event.iScanCode)
    {
        case EStdDeviceKeyFourWayUp:
            newCode = EStdKeyUpArrow;
            break;
        case EStdDeviceKeyFourWayLeft:
            newCode = EStdKeyLeftArrow;
            break;
        case EStdDeviceKeyFourWayRight:
            newCode = EStdKeyRightArrow;
            break;
    }
```

```

    }
    event.iScanCode = newCode;
    iRoidsContainer->OfferKeyEventL(event, aType);
    return CQikViewBase::OfferKeyEventL(aKeyEvent, aType);
}

```

And that's about it. With the view acting as a delegation layer to the Roids container, the game compiles and runs on UIQ with very little effort. The remaining effort was to edit the menu in the resource file to contain an "End Game" item and to ensure that the game paused when focus is lost in the view:

```

void CRoidsUIQView::FocusChanged(TDrawNow aDrawNow)
{
    iRoidsContainer->SetPaused(!IsFocused());
}

```

## 6 Other Observations

### 6.1 Debugging aids

Debugging aids can be huge time savers on even small projects. These are little tools or features which are enabled during development.

#### 6.1.1 Show bounding box

Since polygon collision detection first tests for bounding rectangle intersection, it's useful to be able to visualise it.

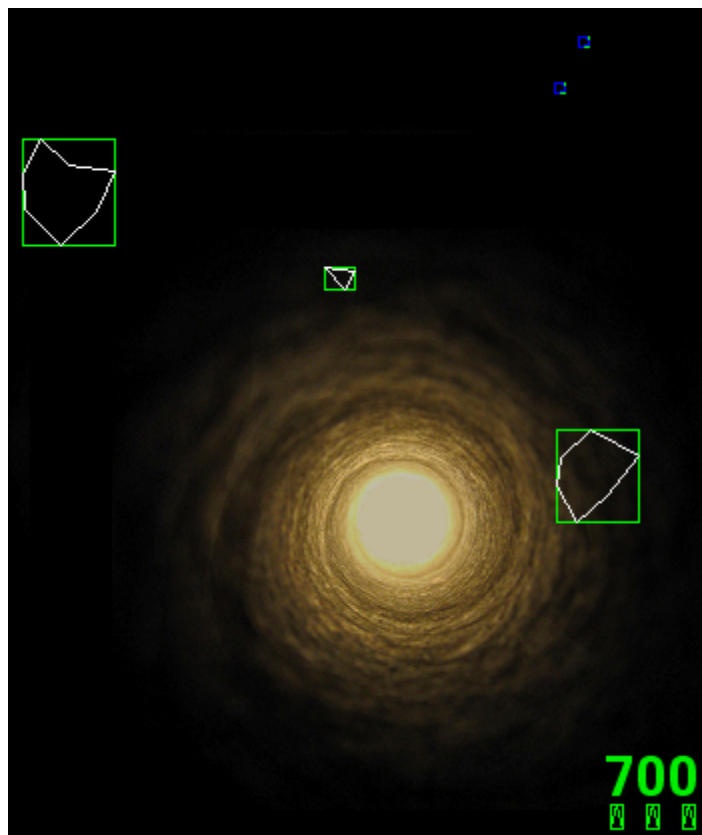
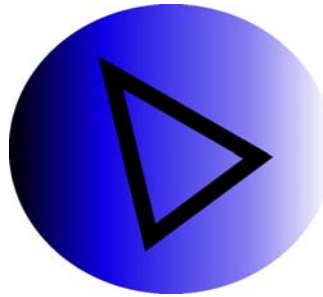


Figure 17 DrawBoundingBox

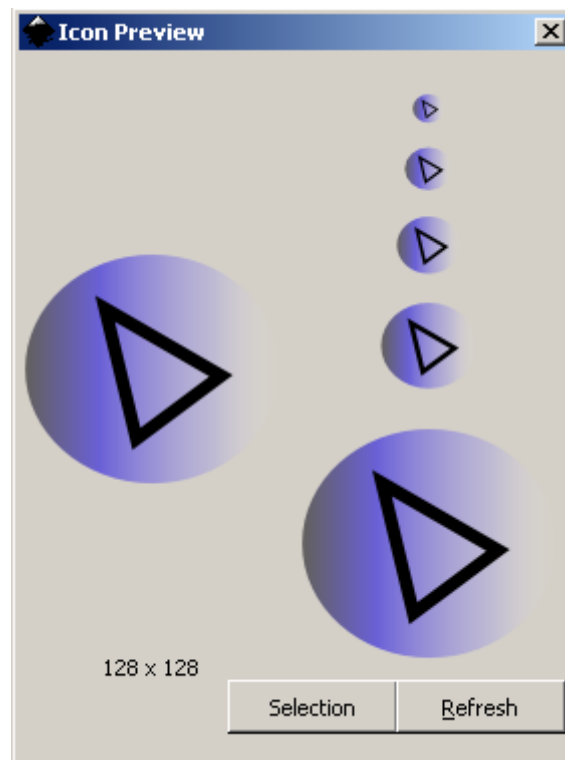
## 6.2 SVG icon design

The icons were created using Inkscape ([www.inkscape.org](http://www.inkscape.org)), an excellent open source SVG-based illustrator.



**Figure 18 Roids SVG icon**

The shape of the ship was copied from the game and layered on a circle filled with a semi transparent gradient. The paper size was set to Icon 32x32 and the handy icon preview was used to test the icon at different sizes



**Figure 19 Inkscape Icon preview**

In order to guarantee compatibility with the S60 build tools, the file is saved from Inkscape in plain SVG format rather than Inkscape SVG format.

## Appendix A - Code to pre-generate sin and cos

This is the code used to generate the lookup table.

```
// sincostables.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <math.h>

const double KPi = 3.141592;

int main(int argc, char* argv[])
{
    printf("/* Automatically generated Lookup table\n");
    printf(" * Generated by sincostables.exe for Roids project\n");
    printf(" * Feel free to use this single source file without restriction */\n\n");

    printf("#ifndef SIN_COS_LOOKUP_DATA\n");
    printf("#define SIN_COS_LOOKUP_DATA\n\n");

    printf("const double KSinLookup[360]={");
    for(int a=0; a<360; a++)
    {
        double sinLookup = sin((double)a * KPi /180.0);
        printf("%f", sinLookup);
        if(a!=360)
            printf(", \n");
    }

    printf("};\n\n");

    printf("const double KCosLookup[360]={");
    for( a=0; a<360; a++)
    {
        double cosLookup = cos((double)a * KPi /180.0);
        printf("%f", cosLookup);
        if(a!=360)
            printf(", \n");
    }
    printf("};\n\n");

    printf("\n\n#endif // SIN_COS_LOOKUP_DATA\n");

    return 0;
}
```

## Appendix B - Exercises

1. The sin/cos lookup table contains duplicate values and part of the table is repeated except it is negative. Can you reduce the size of the lookup table?
2. Add an UFO to the game. The original Asteroids game had an UFO which would occasionally go from one side of the screen to the other. The UFO shoots one shot as it passes. It also scores highly if hit with a projectile.
3. Introduce a new type of asteroid which is made out of a rubbery material. A projectile can split the asteroid as usual but when the ship collides with it, it bounces the ship off. Be sure to bounce off proportionally to the speed of both the ship and the asteroid.
4. Add new features to the capsules. Add a shield feature which causes the ship to bounce off asteroids. Also a type of capsule which, on pick up, causes all asteroids to split at once.
5. Instead of bounding boxes, try using bounding circles as a rough guide to object proximity.

## About the Author



Twm Davies works as an independent consultant in the Symbian sphere. He graduated from Cardiff University with a first in computer science, joining Symbian in 1999 where he worked for almost eight years. He initially worked as a developer of the 'crystal' messaging application which provided the UI to the Nokia communicator range and later specialised in consulting on handset stability and performance, assisting in taking many familiar Symbian devices to market.

Twm is a contributing author to the recently published "Symbian OS Communications Programming" book from Symbian Press and is the author of MobileMind - mind mapping software for S80-based devices (<http://www.twmdesign.co.uk/mobilemind/>).

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.