Sprite Engine

Change History

When	Who	Why
Jan 26, 2000	Gitm	Initial version

Introduction

This document describes the required behaviour and suggested implementation of the sprite and rendering system for TankWar.

Engine Requirements

There are a number of general requirements the engine must meet. The requirements imposed by the TankWar game play are defined in another section.

The engines algorithms should not restrict the implementation to any particular graphics library or colour depth.

By this I mean that the algorithms should be equally applicable to any suitable graphics library (Min 640x480 256 colour mode supported) and should not limit the implementation to any particular colour depth. TankWar is being designed for 16bpp modes using DirectX as the rendering library however we may want to port the system (or parts of it) to other languages/platforms. The algorithms and data structures required by the engine should not impose unnecessary limits on this.

TankWar Game Play

This section describes the expected game play that the rendering engine is expected to be able to form. I've split it into three sections, rendering (generating the screen image), collision detection (this will drive most of the game logic) and sprite movement (which keeps things going in the background).

Rendering

The TankWar game takes place on a variable sized map comprised of a number of pre-defined tiles. The game elements (tanks, bullets etc) are animated on the top of the map. The player can only see a portion of the map at any one time, the area of the map that is in their viewable region is scrolled in the screen window as necessary.

To render the current viewable portion the engine must first generate the portion of the map that is currently visible and then generate any sprite images that fall in that region on top of the map. As shown in (pic), the map forms the bottom layer of the rendered objects – map tiles therefore cannot be transparent so this does not have to be taken into account during that stage of the rendering process.

Tiles are not animated – the renderer does not have to take this into account. The map (which defines the positions of the tiles) is not static, the renderer cannot assume that the map it is rendering from in one invocation will be the same as the map it rendered in the last invocation.

Sprites on the other hand may be irregularly shaped (enclosed in a bounding rectangle) and may have transparent regions. The renderer must be capable of handling this. DirectX supports blitting with a transparent colour – however, this may not be the best solution for our implementation, especially if we want to support multiple colour depths. I discuss the colour depth problem a little later in this document.

Another option is to attach a monochrome mask to each sprite, a black pixel in the mask (binary value 0) would indicate that the corresponding pixel in the sprite image should be rendered as transparent. A white pixel specifies that the corresponding pixel in the sprite should be rendered opaque. This transparency mask could also be used for collision detection – this is discussed shortly. There is no reason why the engine could not use a combination of the transparency mask and the actual image data to generate an image suitable to take advantage of the DirectX chroma keyed blitting functionality.

Sprites can be animated. Each sprite may have a sequence of frames which must be displayed at a certain frame rate – the renderer must be able to determine which frame should currently be used. If a transparency mask is used each frame of the sprite will have a separate corresponding mask associated with it.

A sprite also has a rotation. In the case of TankWar there are 16 separate angles that a sprite can have – rather than generate these images 'on-the-fly' from a single source image it would be better (in terms of graphics quality) to have these images already provided as part of the sprite definition. So a single sprite with 10 animation frames would require 16 x 10 images and the same number of transparency masks. The rendering engine would need to be able to determine which particular image/mask pair should be used for a particular invocation.

Collision Detection

Sprite objects can collide with each other; collision detection between sprites should be performed at a pixel level, rather than simply depending on bounding rectangle intersection.

As well as colliding with each other, sprites can collide with parts of the map. A particular map cell may have some parts that cannot be traversed by sprites (to implement a wall or a building for example). To do this, each map tile image should have an associated monochrome mask that indicates which parts of the image are solid. The mask should be in the same format as the sprite transparency mask.

The collision detection logic would then only need to work with the appropriate masks to determine if a collision has taken place and what object it has taken place with.

Collision detection needs to be performed on all active sprites, whether they are in the visible area of the screen or not. As TankWar is a multiplayer game action may be occurring in a non-visible part of the screen involving other players.

When a collision is detected the engine should generate an event, passing the information about where the collision occurred and what sprites where involved as some sort of callback. This would allow the game logic to be easily implemented as a separate

module, and the intelligence behind the game to be driven by the interaction between the game objects.

Sprite Movement

Most of the operations involving sprite movement should be under the control of the sprite engine, with the ability to be overridden by the game logic where required.

Each sprite will have a location, a direction and a speed. For each iteration of the sprite engine it would update the location based on the direction and speed vector for that sprite. During the collision event the game logic module would have the opportunity to update direction and speed (but not position), and be able to create or delete sprites at that time (for example, if a collision between a bullet and a tank is detected the logic may delete the bullet and tank sprites and create a stationary explosion sprite).

So the sprite structure would look like:

```
typedef struct _SPRITE {
  int iX;        // Current X position (relative to map)
  int iY;        // Current Y position (relative to map)
  int iRot;      // Current rotation
  int iSpeed;    // Current speed
  int iType;     // Type of sprite (reference to sprite images/masks)
  } SPRITE;
```

The iX, iY and iType members would only be settable on creation. iRot and iSpeed could be set or modified at any time during the sprites lifetime.

Game Loop

This section describes how the game loop would be implemented given the information from the previous section. I'm thinking that the game loop should be spread across two separate threads with the sprite engine running in it's own background thread continuously updating the background buffer. The primary thread would be responsible for monitoring user input (and adjusting sprite vectors accordingly) and displaying the background buffer on the primary display surface. How this would work in practice (taking into account network play) still requires some analysis.

The Sprite Engine

To simplify synchronisation between the sprite engine and the primary thread it would probably be better to have two alternating background buffers. Each odd numbered loop of the sprite engine would write to buffer 0, each even numbered loop would write to buffer 1. When the primary thread required access to a buffer to display it would not block the sprite engine (or be blocked by it while rendering occurred).

The sprite engine would run through the following loop during each iteration:

```
// Movement and collision phase
For each active sprite
  Calculate new position based on rotation and speed
  Check for collisions at the new position
Next
// Rendering phase
```

```
Select the next available background buffer
Render the visible map area to the current background buffer
For each sprite in the visible area
  Render sprite to current background buffer
Next
```

One problem with this logic is that $N(N - 1) + N = N^2$ collision checks are required for each sprite (Checking against each other sprite plus a check against the map tiles). It may be possible to check each sprite against all other currently unchecked sprites, this would result in significantly less checks. We need to determine if certain collisions would not be detected by this optimisation and how important they are.

Network Play Considerations

In a networked game, when the application is running as a client the sprite engine does not have to perform the collision detection phase of the game loop. Sprite location information would be sent to the client from the machine acting as the server – all clients share a single sprite engine which is responsible for keeping all other machines updated.

It should also be possible to have a dedicated server, this server would have no user interface and therefore would not require the rendering phase of the operation. A dedicated server should be able to host multiple games.

References

The following references were used in the preparation of this document.