More than SDL

The Hit Producer

SDL provides you with the graphics components to write a game. Unfortunately, there is more to games programming than a graphics engine. This month Steven Goodwin looks at some of things that SDL doesn't do, and demonstrates how we can! BY STEVEN GOODWIN

SDL is a multi-media API. The bulk of its functionality may be held within the graphics subsystem, but it also has libraries for sound, joysticks and threads. However, there is no support for collision detection. This is not surprising as that would fall outside its scope. In the 3D arena, OpenGL, doesn't provide a collision detection framework either. Writing a game without collisions is like eating curry without beer – ultimately unsatisfying!

It's Raining Men

Collision is like the weather – it isn't any one single thing. The collision code to check that the player hasn't walked into a wall is different from that of the player walking into an enemy. And that is different from the collision of the player picking up a key. We shall now cover various methods of collision detection, how they work, why that particular method is pertinant, and where SDL helps us implement it.

Box 1: Projectile considerations

The problem in moving objects several pixels in one frame is called *projectile considerations*. Imagine a bullet moving very fast to the left. Now imagine the player moving very fast to the right. In the real world, the bullet would hit the player. In the game, if we only considered the validity of the end position, it might not!

The bullet could end up completely on the left of the player after a single frame, and no collision would have occurred. To prevent this, we must write special code that checks the entire path that the bullet travels. Because (most of us) are more interested in writing games than mathematical equations, we'll use brute force to check each point along the path.

Moving from position A to position B is not as simple as you'd think. And it certainly isn't as simple as it was when platform games like *Manic Miner* first came out, but the basic principles are the same. We assume that the player will start in a safe (collision-free) location in the level, and require that we will still be in a safe position at the end of each frame.

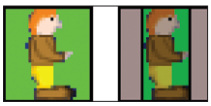

Figure 1: Our character, and his bounding box

If we can't find a new safe position, then we must revert to our original location (assuming this location is still safe), or kill the player. This is game design decision, not a technical one. As games have become more complex, with more and more moving objects, this problem has become more difficult. When true 3D games became available, this complexity increased by several orders of magnitude. Even today, commercial games have to cheat with their collision systems to make them playable on a reasonably modern PC, or console, because the mathematics involved is so complex.

Five to One

For our *movement* collision system we shall check one pixel at a time, looking for collisions, and resetting to the last known safe position if this is not possible. This is the approach taken by most games of this type. Although we *check* one pixel at a time, we shall *move* several pixels over the course of a single frame. This allows us to vary the speed for walking and crawling, and move across the level in a much more realistic manner.

Checking several positions in a frame might appear wasteful, but it is more reliable than just checking that the end point is valid (see Box 1: Projectile considerations) and more flexible than

Listing 1: Collision or not

```
SDL_Rect PlayersBoundingBox;
EX_COLLISION_LIST ListOfCollisions;

  plyGetPlayerRect(pPlyObj, &PlayersBoundingBox);
 exGetRectCollision(pPlyObj, &PlayersBoundingBox, &ListOfCollisions);
```

Listing 2: From *exCheckCollision* in *collision.c*

```
tx = pCollisionRc->x / TheGame.iTileWidth;
 ty = pCollisionRc->y / TheGame.iTileHeight;

 TestTileCollision(pCollisionRc, tx, ty, pCollisionList);
 if (pCollisionRc->x%TheGame.iTileWidth)
  TestTileCollision(pCollisionRc, tx+1, ty, pCollisionList);
 if (pCollisionRc->y%TheGame.iTileHeight)
  TestTileCollision(pCollisionRc, tx, ty+1, pCollisionList);
 if ((pCollisionRc->x%TheGame.iTileWidth) && (pCollision➥
  Rc->y%TheGame.iTileHeight))
  TestTileCollision(pCollisionRc, tx+1, ty+1, pCollisionList);
```

writing specific code to scan the path for obstructions. Our function to do this is called *plyCheckCollisions*.

To determine whether any particular position is safe, we must compare the player's position against every tile in the world to see if they overlap. To simplify this problem, we shall not consider the image data of the player, i.e. his precise outline. We shall only look at the box surrounding him, known as the *bounding box*. If this box collides with the bounding box of any tile, we will be unable to move into this position.

This is not an uncommon trade off, as any problems in the collision code are as annoying for the player to play, as they are for the programmer to program! The player could walk into an alcove, play a "scratch nose" animation, and then find that one particular pixel is in collision with the world, and he can no longer walk out. So, we specify to the artist(s) that the player graphic should remain within a fixed area, and any pixels outside will not be considered during collision detection. For the same reason, we maintain a statically sized bounding box throughout the game.

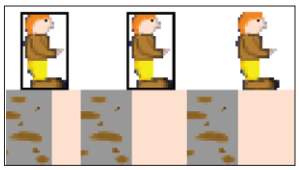

Figure 2: The bounding box does not always reflect the graphics

Our function to check for collision works like this: We start by computing the bounding box of the current player, and then request details of almost every object inside this rectangle (we're hijacking the SDL_Rect structure, even though SDL has no collision support itself). I say *almost* because the first argument to exGetRectCollision is used as an 'ignore' parameter. This is because if we were to check *everything* in the world against the players bounding box there would always be at least one collision – the player! As this is not useful, we explicitly ignore it.

The *ListOfCollisions* will tell us more about our collision state: how many collisions there were, what objects they affected (tiles, enemies, and so on) and even the x,y location where the collision occurred. This latter piece of information would allow us to determine whether an arrow hit us in the head, or the leg, for example. We could then use this to modify the gameplay. By having a general-purpose collision routine such as this, we can focus our efforts in other areas. For the time being, we're only interested in whether our new location is safe, or not.

Inside the *exGetRectCollision* function we can optimize the "check every tile in the world" code to just check the tiles with which we overlap (a speed improvement that will benefit the whole game; another reason for using general-purpose collision code).

exGetRectCollision works in the same way as the 'repair' function we wrote last month. We shall also ignore enemy collisions at this stage, since we perform a different type of collision check on them, and it would be wasteful to perform this test twice.

If we're jumping, for example, and we collide with something we can't just stop in mid-air. We have to consider letting the player fall. Or if we're jumping up and to the left and collide, should we try jumping straight up instead and ignore the left movement, or just start falling? These are all gameplay decisions,

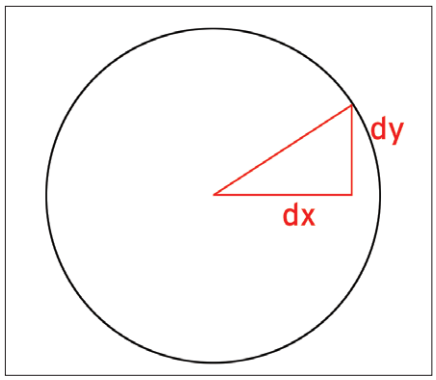

Figure 3: Proximity Checks

and are implemented quite simply by utilizing the same *exGetRectCollision* function for each potential new position. The *plyUpdJumping* function shows which decisions I made for the jumping component.

This same *exGetRectCollision* function can be used for another part of the movement collision – ground detection: we must find out if we're standing on something solid, or not. This can use the same function, but with a different collision rectangle. Principally, a small box just below the feet, that is the width of the player. If there is a collision, we're happy – the player is safely standing on something.

If there isn't, we must switch into a falling state and process our collision in the normal way as the player descends. Ground detection highlights one limitation with the bounding box collision method, as one pixel might collide with a surface, but the graphics do not, as shown in figure 2.

In The Midnight Hour

It is much simpler to detect which objects the player wants to pick up. Although our general-purpose collision routine would work, it's not always the best method, as it is unnecessary to perform any collision detection at all!

Listing 3: School day maths

```
dx = pObj1->x - pObj2->x;
 dy = pObj1->y - pObj2->y;
 iDistance = sqrt( dx*dx + dy*dy
);
 if (iDistance < iProximity)
  { /* ... pick up is in range
... */ }
```

Listing 4: Pythagoras made easy

```
dx = pObj1->x - pObj2->x;
dy = pObj1->y - pObj2->y;
if (iDistanceSquared <
iProximity*iProximity)
   { /* ... pick up is in range
... */ }
```

Listing 5: Collision surface array

```
if (SDL_LockSurface(pCurrentSurface) < 0)
   {
   fprintf(stderr, "Couldn't lock surface: %s\n", SDL_GetError());
   return NULL;
   }
 pCollisionData = (Uint8 *)malloc(pCurrentSurface->w * pCurrentSurface-
>h);
 /* pCurrSurface->pixels now points to valid pixel data for read/writing */
 SDL_UnlockSurface(pCurrentSurface);
```

Pickups are triggered by proximity, so as soon as the player is within, say, half a tile's width (or 16 pixels) we can pick up the object up.

The mathematical way to check for distance is to use Pythagoras: "the square of the hypotenuse is equal to the sum of the squares on the other two sides". This lets us perform a circular check, looking for any object within a specific radius of our current position.

Since the square root is slow and only works on floating point and double precision numbers we shall save time by not doing it! We're not interested in the answer to the equation – only it's result! Is it inside the proximity radius? We can square each side of the equation for the same result.

Space Odditity

Initially, we can use bounding box collision for the player vs enemy collisions, too. However, since we can have a wide range of different enemies (and only one size of tile in which to store them), it might be a little unfair if we introduced fairly small graphics that could kill the player at a distance of 30 pixels!

Similarly, if the animation caused the enemy to squash to half its original size, but we still considered the enemy's collision when at its full height, that would also be unfair. We therefore need a form of collision that is pixel-based – and that requires us to delve back into SDL to determine which pixels are transparent. Unfortunately, it's not as simple as, perhaps, it could be.

Sweetest Perfection

Pixel-based collision is actually quite easy in theory – check each pixel in the image and if any of them are opaque (i.e. not transparent) we have found a collision. SDL, for all its strengths, does not provide a simple GetPixel function. In fact, it does not provide *any* GetPixel function. So we need to write one.

The first stage in the process is to *lock* the surface. Locking is a method where the surface image (whether it is stored in video, or system, memory) is copied to a new buffer in system memory. You can then read, write or change the pixels in this buffer as you need and then, when you unlock the surface, this new data is copied back into the original surface.

While a surface is locked, you can not blit to, or from, it. It is the copying of this data to and from memory that makes locking the surface a very expensive operation, therefore single pixels should never be written individually. If you are working with special bitmap algorithms (like fractals, for instance) then compute a large number of pixels, and write them all into the locked surface together. For collision (which is an exceptional circumstance), we will lock the surface once at the start of the game, copy the data we need into what we'll call our collision surface, and then unlock the surface immediately.

Our collision surface will be a simple array, where one element in the array equates directly to one pixel in the image. A value of zero in the array means 'no collision', whereas a one means 'collision'. For simplicity, each element in the array will be a byte, although in the future we can save memory by reducing this to a single bit.

The pixel data (in *pCurrentSurface->pixels*) is stored in the same format as the surface. As we have already seen (in parts 1 and 2 of this series), each surface may be created with a different bit depth, so we need to understand how to read different pixel formats.

There are two issues here. First off, we need to know what the pixel format is:

```
iBytesPerPixel = ➥
pCurrentSurface->➥
format->BytesPerPixel;
pSurfacePtr = (Uint8 ➥
*)pCurrentSurface->➥
pixels;
```

With 1 byte per pixel, we're using a palletized surface, and so have to consult the palette to get the red, green and blue components. That data is readily available from the surface, as shown in listing 6.

With the packed pixel formats (2,3 or 4 bytes per pixel) we have to perform a little bitwise arithmetic to isolate the individual RGB components, and *normalize* them so that each is within the range 0 to 255. Listing 7 shows this code.

The conversion routine in Listing 7 is identical for all packed formats. The surface has its own combination of masks, shifts and loss parameters to achieve

Listing 6: Consul the palette

```
if (iBytesPerPixel == 1)
  {
  pixel = *(Uint8 *)pSurfacePtr;
  r = pCurrentSurface->format->palette->colors[pixel].r;
  g = pCurrentSurface->format->palette->colors[pixel].g;
  b = pCurrentSurface->format->palette->colors[pixel].b;
  }
```

Listing 7: Normalizing

```
Uint32 pixel;
SDL_PixelFormat *fmt = pCurrentSurface->format;
pixel = *(Uint32 *)pSurfacePtr;   /* Reading from a 4-
bytes-per-pixel surface */
r = ((pixel & fmt->Rmask) >> fmt->Rshift) << fmt->Rloss;
g = ((pixel & fmt->Gmask) >> fmt->Gshift) << fmt->Gloss;
b = ((pixel & fmt->Bmask) >> fmt->Bshift) << fmt->Bloss;
```

Listing 8: Reading bytes

```
if (iBytesPerPixel == 2)   pixel = *(Uint16
*)pSurfacePtr;
if (iBytesPerPixel == 3)   pixel = (*(Uint32
*)pSurfacePtr)&0xffffff;
if (iBytesPerPixel == 4)   pixel = *(Uint32
*)pSurfacePtr;
```

Listing 9: Comparing color

```
if (SDL_MapRGB(pCurr->format, r, g, b) ==
pCurrentSurface->format->colorkey)
  *pCollisionData = 0;   /* this pixel is transparent */
 else
  *pCollisionData = 1; /* this pixel is not! */
 pCollisionData++;
```

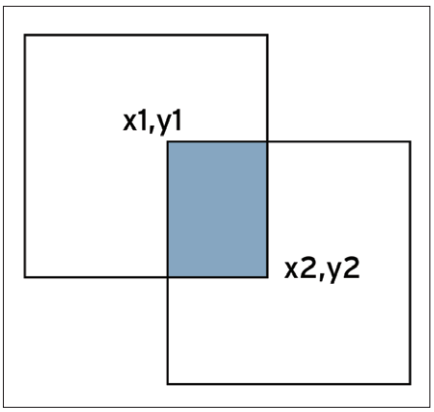

Figure 4: A Union Set

Listing 10: Increasing the pixel pointer

```
pSurfacePtr = (Uint8 *)pCurrentSurface->pixels;
for(y=0; y<pCurrentSurface->h; y++)
 {
 for(x=0; x<pCurrentSurface->w; x++)
  {
  /* ... read the pixel data as above .. */
  pSurfacePtr += iBytesPerPixel;
  }
 /* Move onto the next line */
 pSurfacePtr -= pCurrentSurface->w * iBytesPerPixel;  /* rewind to the
beginning of the line */
 pSurfacePtr += pCurrentSurface->pitch;  /* pitch is already in bytes */
 }
```

this unity. The only additional code (shown in Listing 8), is needed to read the correct number of bytes from the surface pointer.

Having collected the RGB data, we can now use the *SDL_MapRGB* function to create a surface-compatible color and compare it with the color key, writing the result into our collision array.

After reading one pixel, we must move onto the next one. Although this might appear like a simple loop, it is here that the second issue comes to the fore. That is of the *pitch*, or *span*, of the surface.

Singing Meaningless Songs...

When a surface is created, it is not always the size we originally requested. Surfaces in video memory, for example, like to be a power of two (128, 256, 512), and so our 640x480 screen could potentially be 1024x512 pixels. Since the actual size is under control of the graphics card there is very little we can do about it.

When using the standard blitting functions, SDL will automatically consider this for us. But since we've bypassed SDL, we must be aware of what's happening under the hood, since each line may have an extra 384 pixels in it. The *pCurrentSurface->pitch* value holds the size (in bytes) of each line in our surface. We must therefore increase our pixel pointer accordingly. See Listing 10.

Our collision data (because it's ours, stored in our memory) does not have the pitch problem, and so each byte can be stored sequentially. For reference, the full function is called *exCUBuildCollisionSurface*, and located inside *collision.c*.

Wiggle It (just a little bit)

We now have to write our *exCCBoundingBox2Pixel* function, to determine if there are any lethal enemy pixels inside our players bounding box. Again, SDL offers no support for this, and so it is up to us to read through each pixel manually. There are two shortcuts we can take, however. The first is that we can check the bounding boxes of each object first, before we do anything else. If these do not intersect, then the pixel-perfect collision test is a waste of time, and we can exit the function early.

The second optimization is to check only the area of pixels where both bounding boxes overlap, i.e. the union set. SDL doesn't provide any functions to compute this union, but we can do it ourselves fairly simply by considering each case separately: is the enemy above or below the player? Is the enemy to the left or right of the player?

Having found the area to check, (x1,y1) to (x2, y2), we must now find the first pixel in our collision data that pertains to the (x1, y1) position (listing 11), and then iterate through each pixel in turn. Having stipulated a consistent width of 640 for our graphics has an extra bonus here too! Since our collision data relates directly to the graphics surface, every line of collision data is always 640 pixels, which makes it easy to move from one line to the next.

Our collison detection loop (listing 12) has many echoes of the code to create a collision surface, while the computation of x1,y1 borrows code from the *exDrawTile* code – both should be easy to follow. See the *collision.c* file.

One alternative solution is to create an 8-bit surface with *SDL_CreateRGBSurface* and write the collision data into it, perhaps with a blit, or a lock-unlock combination. The surface can then be locked for the entire duration of the game (because nothing will ever blit to, or from, it), in order to save processor time. There's always more than one solution to a problem. As an exercise try [1] and go to the sources/sdl folder. ■

Listing 11: Finding x1 and y1

```
pColData = pCollisionData;
pColData += (iRegion%TheGame.iNumTileWidth) * TheGame.iTileWidth;
pColData += (iRegion/TheGame.iNumTileWidth) * TheGame.iTileHeight * 640;
pColData += x1;
pColData += y1 * 640;
```

Listing 12: Collision detection loop

```
for(y=y1;y<y2;y++)
  {
  for(x=x1;x<x2;x++)
   {
   if (*pColData)
    {
    /* Collision found!! Store
it!!! */
    return TRUE;
    }
   pColData++;
   }
  pColData -= (x2-x1);
  pColData += 640;
  }
```

INFO

[1] *http://www.bluedust.com/pub*