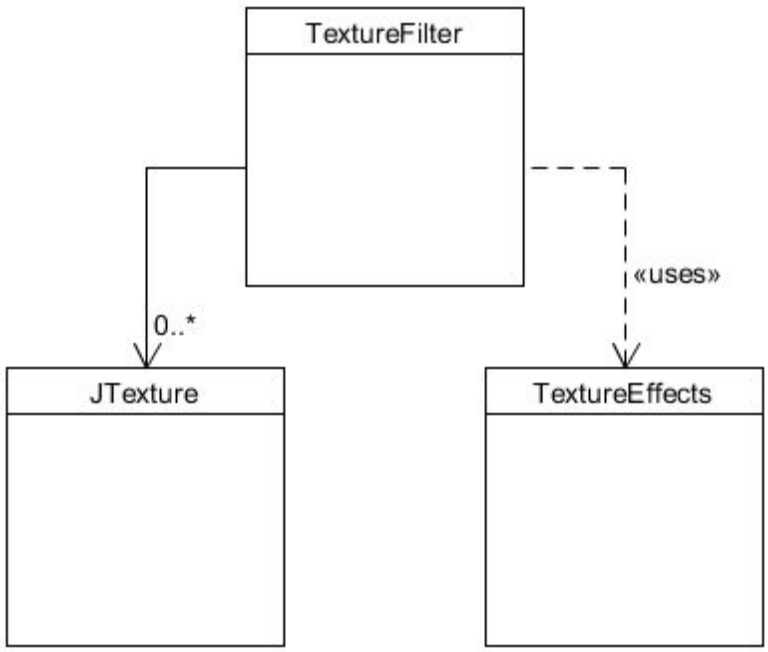
Technical Design Document For RWM P30 2016-17

Class Diagram



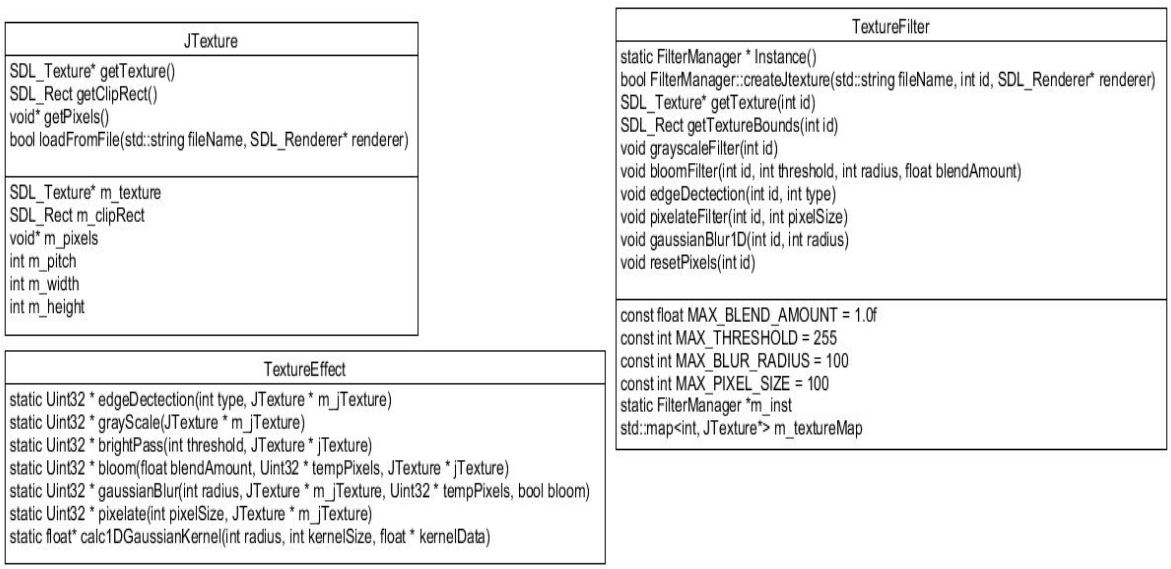
CRC Cards

JTexture	
• Load Images • Create modifiable texture • Get texture pixels • Get texture pitch • Get texture width • Get texture height • Get texture bounds	

TextureEffect	
• Reset texture pixels • Apply blur to texture • Apply grayscale to texture • Apply bloom to texture • Apply blur to texture • Apply edge detect to texture	

TextureFilter	
• Manages JTexture Load/Create • Manages JTexture Get Functions • Manages TextureEffects appy effects • Stores Jtexture Data	• JTexture • TextureEffects

Detailed Class Diagram



Approaches

Two approaches to this project

- Render all textures off screen and apply texture filter to the scene.
- Apply the texture effect to each texture.

Approach #1 - “Tried and tested, example available on request”
The idea is to render all the textures to the scene texture and apply the texture filter to the scene texture. This approach is very costly but low in memory consumption
Pros

- Apply the texture filter once to the scene instead of each texture individually.
- Saves memory as pixel data does not have to be stored.

Cons

- Reading pixels from graphics card is very slow.
- Less control with texture filters per texture.

Approach #2
The idea is to allow each texture to have a filter be applied to it. This approach requires storage of all textures and pixel data. Requires more memory.
Pros

- Getting the pixel data is faster as it will be in memory when loaded compared to getting it from the gpu at run time.
- Apply different filters to different textures.

Cons

- More memory usage from storing pixel data per texture loaded.
- Applying texture filters per texture.

I have decided to go with Approach#2 after building a minimal example using Approach#1. Using the first approach the frames per second maxed out at 40+. This is unacceptable especially before any update code is written and run. The function call to get the Pixels from the Renderer is very expensive as it comes directly from the GPU itself.

The second approach stores the pixels when an image is loaded into a surface so they can be modified at any time without the need to get them from the gpu. It also allows for more control over having multiple filter over different textures rather than applying one filter to the scene texture.

Features

JTexture - “Texture with a few extra bits”

JTexture class is responsible for creating textures that can have its pixel data updated or edited. It is also holds all the data needed in the modification of the textures such as a copy of the textures pixels “Faster to keep a copy then ask the graphics card for them”, height, width and pitch and the bounds of the texture.

```
// JTexture load from file function
bool JTexture::loadFromFile(std::string fileName, SDL_Renderer* renderer)
{
	// SDL_Image load .png
	SDL_Surface* surface = IMG_Load(fileName.c_str());

	// Create new texture to allow editing of pixels
	texture = SDL_CreateTexture(renderer, SDL_PIXELFORMAT_ABGR8888,
	SDL_TEXTUREACCESS_STREAMING, surface->w, surface->h);

	//Lock texture for manipulation
	SDL_LockTexture(texture, NULL, &m_pixels, &m_pitch);

	//Copy loaded/formatted surface pixels
	memcpy(m_pixels, surface->pixels, surface->pitch * surface->h);

	//Unlock texture to update
	SDL_UnlockTexture(texture);
}

// Returns Texture
SDL_Texture* getTexture();

// Returns Texture Bounds
SDL_Rect getClipRect();

// Returns Pixels
void* getPixels();

// Returns Height of Texture in pixels
int getHeight();

// Returns Width of Texture in pixels
int getWidth();

// Returns the Pitch of the Texture - Pitch is the number of bytes between lines
int getPitch();
```

Create TextureEffect

Texture Effect class is responsible for creating and returning the modified pixels which will be used to update the JTexture's modifiable texture. It is also responsible for creating the gaussian kernel used in the blur effect.

Edge Detection

Edge Detection uses 3 preset matrices to produce different edge detection effects

```
// Return the modified pixels that apply an edge detection effect
static Uint32 * edgeDectection(int type, JTexture * jTexture);

// There a 3 types of edge detection effect each using its own matrix
int edge_enhance[3][3] = { { 0,0,0 },{ -1,1,0 },{ 0,0,0 } };
int edge_detect[3][3] = { { -1,-1,-1 },{ -1,8,-1 },{ -1,-1,-1 } };
int emboss[3][3] = { { -2,-1,0 },{ -1,1,1 },{ 0,1,2 } };

// Get Pixel count "Total pixels per texture"
int pixelCount = static_cast<int>(jTexture->getPitch() * 0.25f) * jTexture->getHeight();

// Create a new set of temp pixels "Avoid overwriting the original
Uint32* tempPixels = new Uint32[pixelCount];

// Pointer to the original pixels which are used for reading
Uint32* pixels = (Uint32*)jTexture->getPixels();

// Loop through each pixel in the x and y axis : Jtexture holds the width/x and height/y
// Loop through the matrix in the x and y axis

// Extract the colored data by "Right" bit shifting the pixel data
Uint8 r = pixels[pos] >> 16 & 0xFF;
Uint8 g = pixels[pos] >> 8 & 0xFF;
Uint8 b = pixels[pos] & 0xFF;

// Multiply each color value by the values in the matrix
// Sum up all the pixel colors while looping through the matrix
// Clamp the sum values if they go above 255 or below 0
// Build the pixel data by "Left" bit shifting the color data
tempPixels = Uint8(sumR) << 16 | Uint8(sumG) << 8 | Uint8(sumB);
```

Grey Scale

Grayscale uses preset values to overwrite the current pixels colors with the luma value of that pixel.

```
// Return the modified pixels that apply a grayscale effect
static Uint32 * grayScale(JTexture * m_jTexture);

// Get Pixel count "Total pixels per texture"
int pixelCount = static_cast<int>(jTexture->getPitch() * 0.25f) * jTexture->getHeight();

// Create a new set of temp pixels "Avoid overwriting the original”
Uint32* tempPixels = new Uint32[pixelCount];

// Pointer to the original pixels which are used for reading
Uint32* pixels = (Uint32*)jTexture->getPixels();

// Loop through every pixel
// Extract the colored data by "Right" bit shifting the pixel data
Uint8 r = pixels[i] >> 16 & 0xFF;
Uint8 g = pixels[i] >> 8 & 0xFF;
Uint8 b = pixels[i] & 0xFF;

// Calculate the luma value of the colors
Uint8 v = static_cast<Uint8>((0.212671f * r) + (0.715160f * g) + (0.072169f * b));

// Build the pixel data by "Left" bit shifting the color data
tempPixels[i] = (v << 16) | (v << 8) | v;
```

Bright pass

Bright pass is used in the creation of the bloom effect. Similar to grayscale it calculates the luma value of the pixels. If the luma value is less than the threshold it becomes black, otherwise we retains the pixels color. Only pixels that have color become blurred and are added to the bloom effect

```
// Return the modified pixels that apply the bright pass effect
static Uint32 * brightPass(int threshold, JTexture * jTexture);

// Calculate the luma value of the colors
Uint8 v = static_cast<Uint8>((0.212671f * r) + (0.715160f * g) + (0.072169f * b));

// Checking luma value against threshold
if( v < threshold)
{
   tempPixels[i] = (0 << 16) | (0 << 8) | 0; // Black - 0
}
else
   tempPixels[i] = (r << 16) | (g << 8) | b;
```

Bloom

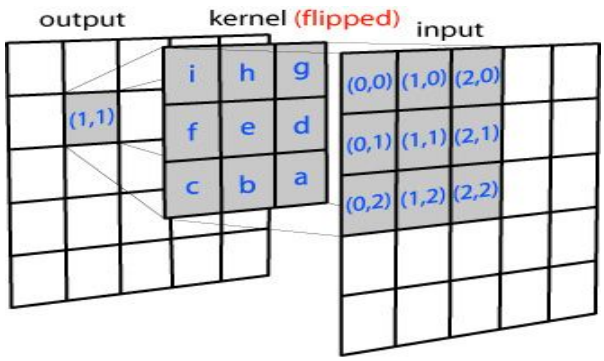
Bloom is the only function that takes an array of modified pixels in as a parameter. The reason for this is the modified pixels get added to the original pixels to create the effect. The modified pixels first go through a bright pass filter and then the blur filter. How much of the original pixels get added to original pixels is based on the blend amount parameter that is also passed to the function.

```
// Return the modified pixels that apply the bloom effect
static Uint32 * bloom(float blendAmount, Uint32 * tempPixels, JTexture * jTexture);

// Get Pixel count "Total pixels per texture"
int pixelCount = static_cast<int>(jTexture->getPitch() * 0.25f) * jTexture->getHeight();

// Pointer to the original pixels which are used for reading
Uint32* pixels = (Uint32*)jTexture->getPixels();

// Loop through every pixel
// Extract the colored data by "Right" bit shifting the pixel data : Modified Pixels
Uint8 r1 = tempPixels[i] >> 16 & 0xFF;
Uint8 g1 = tempPixels[i] >> 8 & 0xFF;
Uint8 b1 = tempPixels[i] & 0xFF;

// Extract the colored data by "Right" bit shifting the pixel data : Original Pixels
Uint8 r = pixels[i] >> 16 & 0xFF;
Uint8 g = pixels[i] >> 8 & 0xFF;
Uint8 b = pixels[i] & 0xFF;

// Add pixels with (tempPixels * blend) which is passed as a parameter
sumR = r + ((r1 * blendAmount));
sumG = g + ((g1 * blendAmount));
sumB = b + ((b1 * blendAmount));

// Clamp any value that is greater than 255.
// Build the pixel data by "Left" bit shifting the color data
tempPixels[i] = (v << 16) | (v << 8) | v;
```

Calculate Gaussian Kernel

In this component the Kernel is a 1D array and the blur effect is applied first horizontally and then vertically. This results in less multiplications and a faster completion time.

Calculate gaussian kernel is a function that creates a matrix of varying sizes whose values summed together add up to 1. The size of the kernel is (2 * radius) + 1. This ensures an odd number sized matrix and guarantees balance when blurring the pixels. The current pixel will always use the centre value of the matrix and will have an equal number of values above, below, left and right of it.

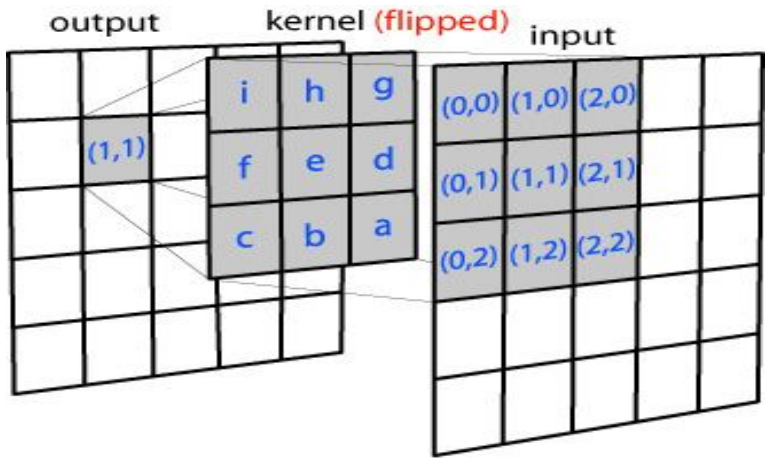



```
// Return array of floating point values
static float * calc1DGaussianKernel(int radius, int kernelSize, float * kernelData);


// Formula for calculating the gaussian kernel
```

$$\left(G(x\,y) = \frac{1}{2\,\pi\,\sigma^2}\,e - \frac{x^2 + y^2}{2\,\sigma^2}\right)$$

Blur

Blur effect uses the gaussian kernel to determine what the color the pixel should. Using **e.g.1** as an example we match up the centre of the kernel with the current pixel that is been edited and multiply the color value by the kernel value. The surrounding pixels are also matched up with the kernel and multiplied by the corresponding kernel values. The addition of these multiplications is the color the current pixel will receive.

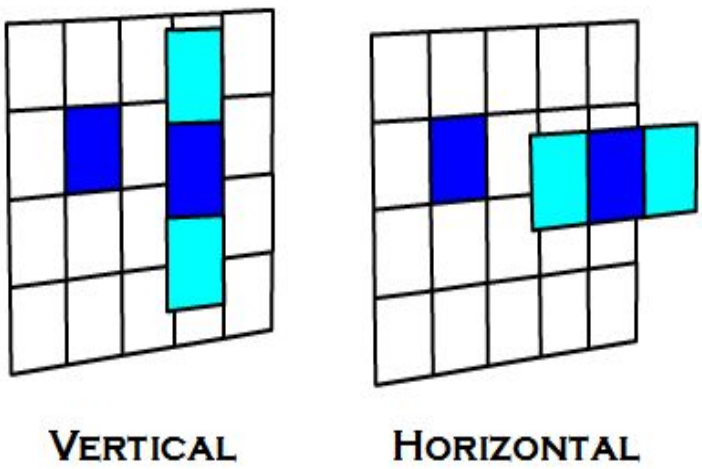


e.g.1

In this example we use a 1D array of kernel values to reduce the multiplication needed. Using **e.g.2** as an example we can achieve the same effect as above by doing 2 passes, one vertical and one horizontal. With the vertical pass we use the above and below pixels from the current pixel and with horizontal we use the left and right pixels of the current pixel. If the overlaid kernel has pixels out of bounds they are wrapped around to the opposite side of the image.

Example pixel[-1][0] = pixel [max_width][0]

Example pixel[0][-1] = pixel [0][max_height]



e.g.2

Below is an example of how many multiplications are saved by using 2 pass blur vs 1 pass blur. This example uses the max blur radius of 50 “Size = Radius * 2 + 1” which results in an array of 101 or a matrix of 101 * 101

Image size
800 * 600 = 480000

Max size Matrix - 101 * 101
480000 * 101 * 101 = 4,896,480,000 Multiplications

Max size array - 101
480000 * 101 * 2 = 96,960,000 Multiplications

4,799,520,000 Multiplications saved.
“This does not include the multiplications for each R,B,G,A value”

```
// Return the modified pixels that apply a blur effect
// temp pixels and bloom parameters are only used when the bloom effect is applied
// otherwise Null and false are passed as the blurs parameters
static Uint32 * gaussianBlur(int radius, JTexture * m_jTexture, Uint32 * tempPixels,
bool bloom);

// Get Pixel count "Total pixels per texture"
int pixelCount = static_cast<int>(jTexture->getPitch() * 0.25f) * jTexture->getHeight();

// Create a new set of temp pixels "Avoid overwriting the original”
Uint32* tempPixels = new Uint32[pixelCount];

// Pointer to the original pixels which are used for reading
Uint32* pixels = (Uint32*)jTexture->getPixels();


// Vertical Blur
// Loop through each pixel in the x and y axis : Jtexture holds the width/x and height/y
// Loop through the array of kernel data
// Multiply each color value by the values in the array
// Sum up all the pixel colors while looping through the array
// Build the pixel data by "Left" bit shifting the color data
tempPixels[i] = (v << 16) | (v << 8) | v;


// Horizontal Blur
// Repeat the above
```

Pixelated

Pixelate effect simulates the reduction of pixels in an image by replacing blocks of pixels with multiple colors with a block of pixels that averages the colors used to form the block. An example of an 800 x 600 image using pixel size 10 would result in a image with 80 x 60 blocks of color.

```
// Return the modified pixels that apply a grayscale effect
static Uint32 * pixelate(JTexture * m_jTexture);

// Get Pixel count "Total pixels per texture"
int pixelCount = static_cast<int>(jTexture->getPitch() * 0.25f) * jTexture->getHeight();

// Create a new set of temp pixels "Avoid overwriting the original”
Uint32* tempPixels = new Uint32[pixelCount];

// Pointer to the original pixels which are used for reading
Uint32* pixels = (Uint32*)jTexture->getPixels();

// Loop through each pixel in the x and y axis : Increment by Pixel Size
// Loop through each pixel size in both x and y axis
// Sum up all the pixel colors while looping through the pixel size loops
// Keep track of the number of pixel size loop increments
Uint8 r = pixels[pos] >> 16 & 0xFF;
Uint8 g = pixels[pos] >> 8 & 0xFF;
Uint8 b = pixels[pos] & 0xFF;
totR += r;
totG += g;
totB += b;
sumNum++;

// Divide the current pixel color by the No. of increments to get the average color.
totR /= sumNum;
totG /= sumNum;
totB /= sumNum;

// Loop through the pixel size in x and y axis and replace all with the average color.
```

TextureFilters

This class is responsible for for accessing / handling the functions for both the JTexture and Texture Effect class. Texture Filter is a singleton class that stores JTexures in a map, ready to be passed as parameters to the Texture Effect class.

```
-------------------------------------------------

// Loads an Image and creates a Jtexture and stores it in a map with ID as the Key
bool FilterManager::createJtexture(std::string fileName, int id, SDL_Renderer* renderer)
-------------------------------------------------

// Grayscale filter takes an id to the JTexture in the map as its parameters
void FilterManager::grayscaleFilter(int id)
{
    // Get grayscale pixels from TextureEffects
    Uint32* tempPixels = TextureEffect::grayScale(m_textureMap[id]);

    // Update Texture with the new pixels
}
-------------------------------------------------

// Bloom filter takes an id to the JTexture in the map, threshold, blur radius
   and blending amount as its parameters
// Bloom uses 2 extra filters in order to achieve its effect, Brightpass and Blur
void FilterManager::bloomFilter(int id, int threshold, int radius, float blendAmount)
{
   // Bright pass Filter
   Uint32* tempPixels = TextureEffect::brightPass(threshold, m_textureMap[id]);

   // Blur Filter
   tempPixels = TextureEffect::gaussianBlur(radius, m_textureMap[id], tempPixels, true);

   // Bloom Filter - Adds Original pixels to modified pixels
   tempPixels = TextureEffect::bloom(blendAmount, tempPixels, m_textureMap[id]);

   // Updates Texture with the new pixels
}
-------------------------------------------------

// Edge filter takes an id to the JTexture in the map, and a type as its parameters
// Type represents which edge filter, “Emboss, Detect and Enhance”
void FilterManager::edgeDectection(int id, int type)
{
   // Get edge detection pixels from TextureEffects
   Uint32* tempPixels = TextureEffect::edgeDectection(type, m_textureMap[id]);

   // Updates Texture with the new pixels
}
```

```
-------------------------------------------------

// Pixelate filter takes an id to the JTexture in the map, and a pixel size as its
parameters
void FilterManager::pixelateFilter(int id, int pixelSize)
{
    // Get pixelated pixels from TextureEffects
    Uint32* tempPixels = TextureEffect::pixelate(pixelSize, m_textureMap[id]);

    // Updates Texture with the new pixels
}
-------------------------------------------------

// Pixelate filter takes an id to the JTexture in the map, and a blur radius as its
parameters
void FilterManager::gaussianBlur1D(int id, int radius)
{
    // Get blur pixels from TextureEffects
    // Blur take 2 extra parameters, these are only ever used if the blur
       is used for the bloom effect. "See Bloom for parameters"
   tempPixels = TextureEffect::gaussianBlur(radius, m_textureMap[id], NULL, false);
    // Updates Texture with the new pixels
}

-------------------------------------------------

// ResetPixels filter takes an id to the JTexture in the map as its parameters
// It resets the texture back to its original state
void FilterManager::resetPixels(int id)
{
    // Get pixels from Jtexture
    Uint32* pixels = (Uint32*)m_textureMap[id]->getPixels();
    // Update current texture with original pixel
}
-------------------------------------------------
```