4
Técnicas de Renderização HDR

Após a apresentação, nos capítulos anteriores, dos conceitos fundamentais e dos trabalhos sobre HDR que influenciaram esta dissertação, esse capítulo tem como objetivo a introdução dos conceitos mais importantes de HDR para renderização em tempo real e a utilização de mapas de iluminação HDR. O *pipeline* tradicional para renderização em tempo real é apresentado. Esse capítulo apresenta os passos fundamentais para a criação de aplicações em tempo real que geram imagens HDR. As aplicações principais dessa dissertação, descritas no Capítulo 5, se utilizam desses fundamentos.

4.1
Superfícies HDR

A renderização de cenários utilizando *HDR images* necessita suporte do hardware gráfico. Tipicamente, os recursos HDR necessários se tornaram disponíveis com a geração de placas gráficas NV40 (GeForce 6800) e R400 (ATi Radeon X800). O requisito primordial para renderizações HDR é a utilização de superfícies (*render targets*) de alta precisão. As tradicionais de renderização, conhecidas como *backbuffers* são inerentemente *Low Dynamic Range* e atualmente é comum apresentarem um formato de 32 bits por pixel, 8 bits para cada canal de cor R, G, B e 8 bits para o canal *alpha*. Cada um dos 8 bits representam um valor inteiro com domínio [0,255].

A utilização dessas superfícies para renderização, devido a sua baixa precisão interna, faz com que os valores dos pixels sejam aproximados durante o cálculo de iluminação e, de forma mais perceptível, satura os valores dos pixels em áreas super expostas. Para evitar a saturação dos valores de pixels, são utilizadas superfícies de alta precisão. Os formatos padrão para tais superfícies são:

– FP16 - A16B16G16R16F: Formato que apresenta 16 bits em representação de ponto flutuante para cada canal. Esse formato foi

o primeiro a permitir renderização HDR verdadeira (com valores de pixels não saturados em 1) sem necessidade de utilizar múltiplos *render targets*. Atualmente esse é o formato mais utilizado pela família de placas da Nvidia, pois necessita apenas de 64bits para representar um pixel, com boa precisão nos cálculos de iluminação.

– FP32 - A32B32G32R32F: Formato que apresenta 32 bits em representação de ponto flutuante para cada canal. Esse formato necessita quatro vezes mais de espaço de armazenagem do que imagens LDR. O suporte a *blending* não está disponível, inviabilizando algumas operações. É importante salientar que algumas placas gráficas capazes de suportar renderizações em superfícies de alta precisão não suportam operações de *color blending* nessas superfícies.

– R16G16F: Formato que apresenta 2 canais de 16 bits com precisão em ponto flutunte para cada canal. Apesar da alta precisão não consegue representar os canais de cor necessários para renderização final, porém pode ser combinado utilizando dois *render targets* para representar os canais de cor e *alpha*. Em algumas placas esse formato é mais eficiente em relação a cache de placa gráfica[12].

– A16B16G16R16: Formato que apresenta 16 bits em representação de ponto fixo para cada canal. Apesar da quantidade maior de bits para cada canal em relação aos formatos de 8 bits, não possui a mesma capacidade de representação interna dos formatos em ponto flutuante. É o formato utilizado pela ATi.

Existem ainda outros formatos de superfícies, como o tradicional formato HILO, que apresenta 2 canais de 16 bits com representação inteira para cada pixel. Esse formato, por não ser ponto flutuante, não apresenta resultados visuais tão bons quanto as texturas em ponto flutuante. Um outro formato utilizado é o R16G16F que apresenta dois canais de 16 bits em ponto flutuante. Algumas placas suportam esse formato melhor que os formatos anunciados acima, porém eles necessitam de decodificação e múltiplos *render targets* para representar os canais de cor e alpha.

4.2 Pipeline HDR

O *pipeline* para renderização utilizando imagens HDR permite que diferentes efeitos possam ser aplicados, gerando imagens com maior realismo que as possíveis anteriormente. Apesar disso, como as superfícies utilizadas

para renderização apresentam uma maior quantidade de bits necessários para representar um pixel, problemas com a taxa de preenchimento (*fill rate*)podem ocorrer, levando a uma queda na performance da aplicações. O estudo do *pipeline* e técnicas para diminuir o impacto do desempenho das aplicações são apresentados a seguir.

O *pipeline* para renderização HDR em tempo real é tipicamente definido pelo seguintes passos:

- Renderizar cena em superfície de alta precisão
- Diminuir tamanho da superfície em 1/4
- Aplicar *bright pass filter*
- Aplicar processamento de buffer (efeitos de *glare*)
- Combinar imagem processada com a cena original
- Aplicar *tonemapping*

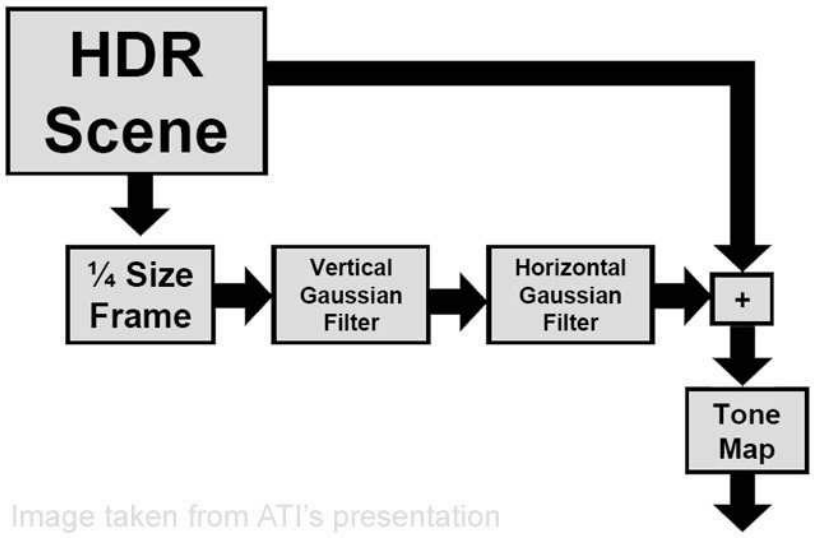

Figura 4.1: *Pipeline* HDR para renderização em tempo real

4.2.1
Renderização da cena

Nesse estágio a cena é renderizada para uma superfície de alta resolução. A interação dos materiais associados aos objetos da cena, a geometria dos objetos e as luzes do cenário gera a imagem base da renderização, que será posteriormente pós-processada. Como a superfície possui alta precisão, valores de iluminação superiores a 1 ficaram registrados no *color buffer*. Nessa fase, um conjunto de técnicas visando a geração de resultados mais realistas, como *Normal Mapping*[22] e *Parallax Mapping*, podem ser utilizados.

4.2.2
Diminuição da superfície de renderização

A diminuição da superfície de renderização é uma tarefa importante para os estágios seguintes do *pipeline*, pois permite uma maior performance das aplicações. A utilização de imagens com menor resolução diminui o tempo necessário para a aplicação dos filtros de processamento. Além disso na composição o *stretching* causado pelo filtro de magnificação ajuda na composição da cena pós-processada ao ser combinada com a imagem original, pois aumenta o efeito de borramento da imagem processada. O código em linguagem HLSL para o passo de escala é apresentado a seguir. *Vertex shader* do passo de escala

```
float4x4 matViewProjection;

struct VS_INPUT
{
    float3 Pos:        POSITION;
};

struct VS_OUTPUT
{
    float4 Pos:         POSITION;
    float2 UV : TEXCOORD0;
};

VS_OUTPUT vs_main( VS_INPUT Input )
{
   VS_OUTPUT Output;
```

```
    Output.Pos.xy = sign(Input.Pos);
    Output.Pos.z = 1.0;
    Output.Pos.w = 1.0;

    Output.UV.x = Output.Pos.x * 0.5 + 0.5;
    Output.UV.y = (1.0 - ( Output.Pos.y * 0.5 + 0.5 ));

    return Output;
}
```

O *vertex shader* tem apenas a função de transformar os vértices em um plano alinhado com a tela (*axis aligned quad*) e passar as coordenadas de texturas do polígono.

Pixel shader do passo de escala

```
sampler2D RT;

struct PS_INPUT
{
    float2 UV : TEXCOORD0;
};

struct PS_OUTPUT
{
    float4 Color : COLOR;
};

PS_OUTPUT ps_main( PS_INPUT In )
{
    PS_OUTPUT Output;

    float4 texColor = tex2D(RT, In.UV);

    Output.Color = texColor;

    return Output;
}
```

A função principal do pixel shader é acessar o valor da textura que contém o *render target* anterior. Para atingir uma melhor aparência para o *render*

Dimensões(pixels)	Desempenho(FPS)
800x600	4
128x128	60

Tabela 4.1: Desempenho baseado nas dimensões do *render target*

target diminuido (fato interessante caso efeitos de pós-processamento não sejam utilizados posteriormente), outras formas de acesso ao pixel podem ser feitas como filtragem linear dos 4 pixels adjacentes. Tal fato é necessário pois alguns formatos HDR não são filtrados autmaticamente pelas placas gráficas. É importante notar também que o estágio de supressão de cores pode ser (e tipicamente é) realizado durante a diminuição do *render target original*, porém para uma melhor explicação, é mantido separadamente. A utilização de *render targets* reduzidos permite que as aplicações baseadas no pipeline tenham um om desempenho. A tabela 4.1 mostra os resultados. Ao utilizar o tamanho do *render target* original para o processamento posterior, a aplicação perde as características de tempo real.

4.2.3 Estágio de Bright Pass Filter

A Iluminação do mundo real apresenta diferentes características, que se aplicadas na renderização de cenas, aumentam seu realismo. Como dito anteriormente, alguns desses efeitos são baseados no fato da luz aparentar vazar para dentro de objetos e luzes pontuais ocuparem uma área maior. Uma particular características apresentada por diferentes ambientes é a concentração das fontes de iluminação. Ou seja, apesar da luz ser absorvida, transmitida e refletida pelos objetos, é possível identificar as fontes de iluminação facilmente, uma vez que seu *dynamic range* é geralmente muito maior que outras partes da cena[47]. É desejável que apenas as áreas da imagem que apresentem altos valores de radiância sofram a aplicação dos efeitos de iluminação. Para tal, é necessário que os pixels da cena com maiores valores sejam separados do resto da cena.

O estágio de *bright pass filter* é responsável pela detecção das regiões mais iluminadas da imagem. Um valor de limiar pode ser utilizado para definir quais pixels devem ser separados do restante da imagem. Para tal tarefa, diferentes abordagens podem ser utilizadas, desde uma simples escolha de selecionar pixels com valores maiores que 1.0 (valores HDR por definição), até a utilização do cálculo de luminância média da cena para comparação com os valores de luminância individuais. Outras técnicas

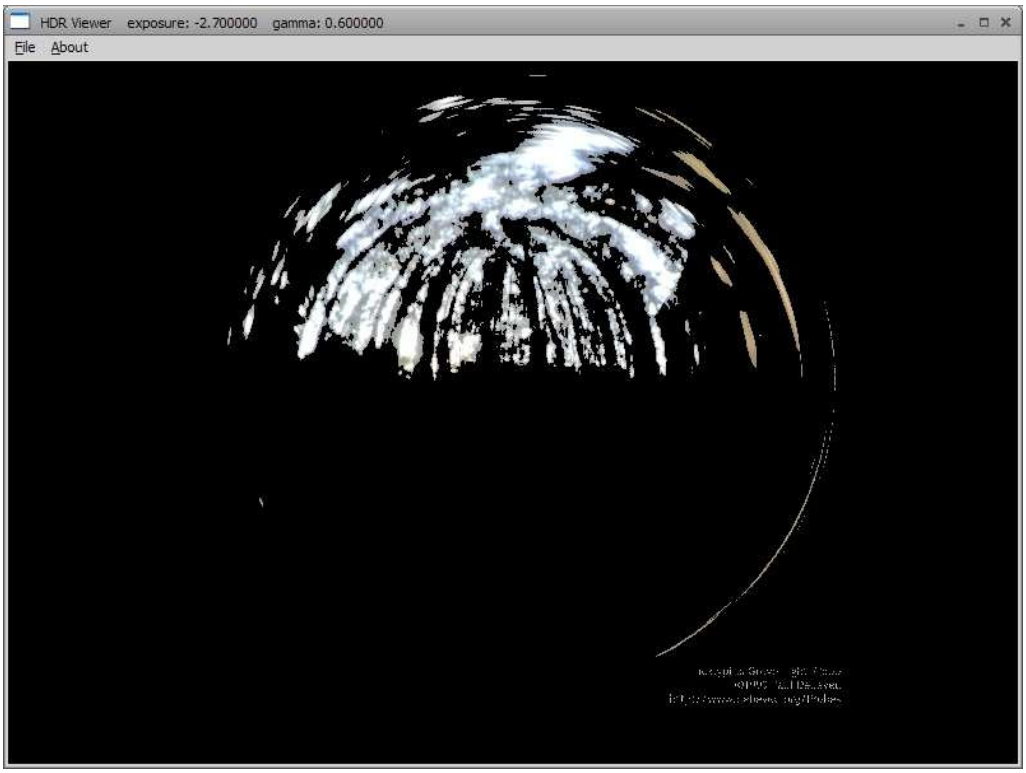

Figura 4.2: Estágio de *bright pass filter*

comuns para mascarar de partes de cenas em superfícies LDR também podem ser utilizadas, entre elas se destaca a utilização do canal alpha das imagens. A técnica consiste em utilizar o canal alpha para codificar informações de luminância dos pixels. Dessa forma, o canal alpha contém a informação de luminância da cena, ou seja, quais pixels são brilhantes e devem sofrer processamente. [13] utiliza a seguinte fórmula para definir áreas mais luminosas baseadas no canal alpha.

$$Lum = \left(\frac{1 - \epsilon}{16} + \alpha \right) \tag{4-1}$$

Onde ϵ é o valor de limiar para seleção de pixels e *alpha* é o valor do canal alpha do pixel. Após esse cálculo basta modular o valor dos canais de cor de pixel pelo valor de luminância, para gerar a imagem com as fontes de brilho coloridas.

$$Pixel = Pixel.RGB * \left(\frac{1 - \epsilon}{16} + \alpha \right) \tag{4-2}$$

Com a modulação dos valores dos canais de cor pelo canal alpha, áreas especificas da imagem podem ser definidas, permitindo um controle prévio das áreas que devem apresentar efeitos de super exposição. Essa técnica

permite um controle de visibilidade de cada pixel da imagem. Se, um pixel específico não deve ser exibido, basta definir o valor 0 para o canal alpha do pixel. Ao modular os valores dos canais rgb desse por seu canal alpha, é fácil verificar que o valor do pixel será 0. Essa técnica é importante também para a utilização de texturas em objetos emissivos, o canal alpha indica que determinado objeto é emissor de luz e consequentemente deve ser uma fonte de *glare*. Através da simples equação

$$Lum = Texel.rgb * Texel.a * \delta \tag{4-3}$$

Onde δ é um valor de escala que permite que o valor do pixel seja superior a 1, o suficiente para geração de *glare* nessa região.

4.2.4 Glare

A iluminação do mundo real apresenta comportamentos interessantes e cuja reprodução em cenários renderizados aumenta o realismo de tais cenas. Um conjunto de efeitos que apresenta particular atrativo são os efeitos baseados em super exposição de determinadas regiões da cena. Nesses casos a luz concentrada aparenta ocupar uma área maior que a área de onde a luz é proveniente. Esses efeitos são comuns também em fotografias, onde o movimento da câmera, movimento dos objetos, desfoco da imagem, e o desvio da luz pelo ar e dentro do sensor.

A aplicação desses efeitos é particularmente interessante para superfícies que conseguem guardar valores de iluminação em alta precisão. Ao utilizar superfícies de renderização que saturam os valores máximos de brilho em 1, informações importantes são perdidas e alteram o resultado final da cena. Considere o exemplo da imagem 4.3. Nela dois quadrados são renderizados com os valores de iluminação 1.5 e 6, equanto o funco da imagem apresenta apenas pixels com valor zero. Devido a não aplicação de um algoritmo de *tonemapping* mais sofisticado, os valores foram saturados em 1 para exibição. Ao aplicar um filtro gaussiano aos valores HDR da superfície, os quadrados irão aparentar ser bastante diferentes, o quadrado a esquerda se torna uma região consideravelmente mais escura que o a direita, efeito condizente com a variação de iluminação apresentada pelos dois quadrados. O quadrado a direita além de gerar uma região mais clara, apresenta um raio de ação mais amplo. Caso se utilize uma superfície LDR para armazenagem dos valores de iluminação, os valores dos pixels seriam

saturados em 1, e ao aplicar o filtro, os resultados seriam iguais para os dois quadrados, claramente um erro.

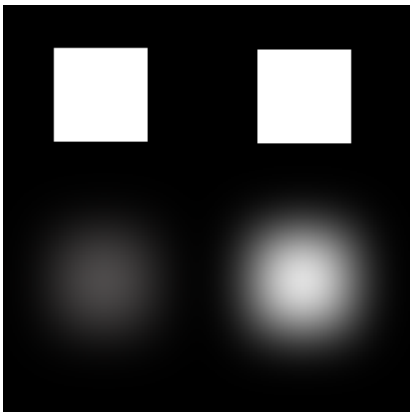

Figura 4.3: Comparação de filtros gaussianos com valores de pixel não saturados

Para atingir um maior nível de qualidade de filtragem é necessária a utilização de diferentes filtros baseados em gaussianas com raios distintos. O raio de ação de um gaussiana é definido pelo coeficiente de desvio padrão $\sigma = \sqrt{x^2 + y^2}$.

$$f(\sigma) = e^{-\sigma^2} \tag{4-4}$$

Quanto maior o raio, mais borrada a imagem resultante será. A plotagem de uma curva gaussiana é apresentada na Figura 4.4

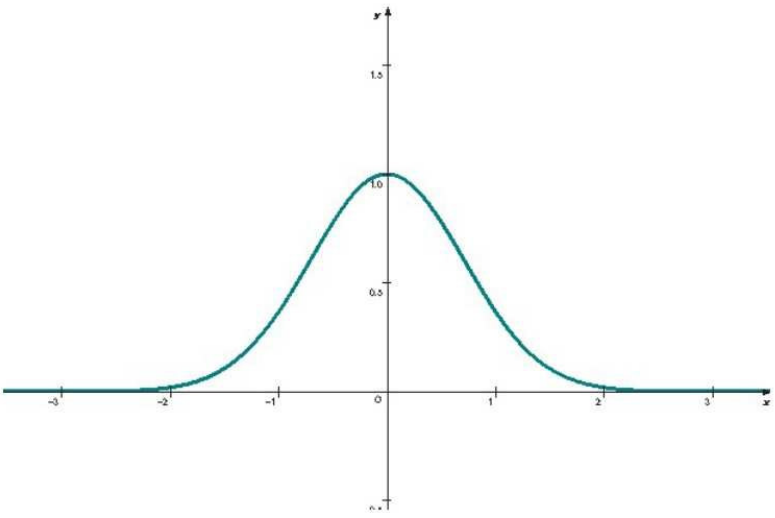

Figura 4.4: Plotagem de curva gaussiana

O processo de convolução de imagens é normalmente computacionalmente pesado, e dependente da resolução da imagem. Porém ao se utilizar algumas técnicas esse custo pode ser diminuído

consideravelmente, a ponto de permitir taxas de renderização em tempo real.

A primeira importante otimização se deve ao fato da separabilidade de filtros gaussianos bidimensionais, ou seja, filtros gaussianos bidimensionais podem ser aproximados através de um filtro unidimensional na horizontal seguido de outro filtro unidimensional na vertical. Filtros gaussianos 1D são computacionalmente mais leves e mesmo combinados em duas passadas, o tempo para aplicação da convolução é menor. O código em linguagem HLSL para o filtro horizontal é representado abaixo.

Vertex shader do filtro gaussiano

```
float4x4 matViewProjection;

struct VS_INPUT
{
    float3 Pos:        POSITION;

};

struct VS_OUTPUT
{
    float4 Pos:         POSITION;
    float2 UV :         TEXCOORD0;
};

VS_OUTPUT vs_main( VS_INPUT Input )
{
    VS_OUTPUT Output;

    Output.Pos.xy = sign(Input.Pos);
    Output.Pos.z = 0.0;
    Output.Pos.w = 1.0;

    Output.UV.x = Output.Pos.x * 0.5 + 0.5;
    Output.UV.y = 1.0 - (Output.Pos.y * 0.5 + 0.5);



    return Output;
}
```

O *vertex shader* apenas transforma os vértices para representar um plano alinhado como plano de visão, e passa as coordenadas de textura para o *pixel shader*, onde a convolução será calculada.

```
sampler2D Src;

float4 gaussianFilter[7] =
{
  -3.0, 0.0, 0.0,  1.0/64.0,
  -2.0, 0.0, 0.0,  6.0/64.0,
  -1.0, 0.0, 0.0, 15.0/64.0,
   0.0, 0.0, 0.0, 20.0/64.0,
   1.0, 0.0, 0.0, 15.0/64.0,
   2.0, 0.0, 0.0,  6.0/64.0,
   3.0, 0.0, 0.0,  1.0/64.0
 };

float texelOffset = 1.0/128.0;


struct PS_INPUT
{
   float2 UV : TEXCOORD0;
};

struct PS_OUTPUT
{
   float4 Color : COLOR;
};

PS_OUTPUT ps_main( PS_INPUT Input )
{
   PS_OUTPUT Output;

   float4 color = 0.0;

   int i;
   for (i=0;i<7;i++)
   {
      color += tex2D(Src,float2
```

```
        (Input.UV.x + gaussianFilter[i].x
        * texelOffset,Input.UV.y + gaussianFilter[i].y
        * texelOffset)
        ) * gaussianFilter[i].w;
    }

    Output.Color = color * 4.0;

    return Output;
}
```

Inicialmente os pesos para cada localidade de texel adjacente são definidos. A configuração dos pesos é exemplificada na Figura 4.5. Após, é declarado *offset* do texel, ou seja, qual a variação do valor para as coordenadas de textura de texels adjacentes. Nesse caso, ao utilizar uma textura com largura de 128 pixels, com coordenadas de textura normalizadas em [0..1],a variação é de 1/128 para as coordenadas de texels vizinhos. O próximo passo calcula o valor final a ser atribuído ao pixel.

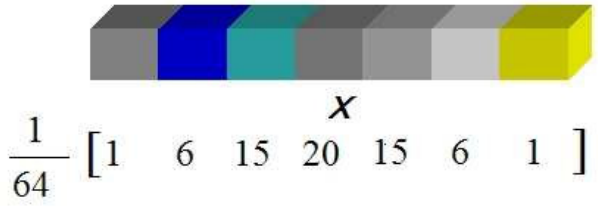

Figura 4.5: Acesso as coordenadas de textura e seus respectivos pesos.

A segunda otimização diz respeito a forma como gaussianas de diferentes raios são utilizadas para gerar um resultado final mais refinado. Utilizar filtros de diferentes raios é computacionalmente pesado, uma vez que o custo computacional das convoluções é definido pela resolução da imagem. Como as imagens geralmente apresentam altas resoluções, o tempo final de computação tende a ser alto.

Para solucionar esse problema, é comum utilizar uma técnica baseada em imagens com diferentes resoluções. Ao diminuir a resolução de uma imagem e aplicar um filtro com o mesmo raio, o resultado atingido é praticamente idêntico ao atingido com filtros de diferentes raios. Como as imagens apresentam resoluções cada vez menores, o tempo para seu cálculo é diminuído proporcionalmente ao inverso do quadrado da resolução original. A figura 4.7 mostra o efeito da filtragem de uma imagem por gaussianas de diferentes raios. A figura 4.8 mostra a combinação de imagens de diferentes

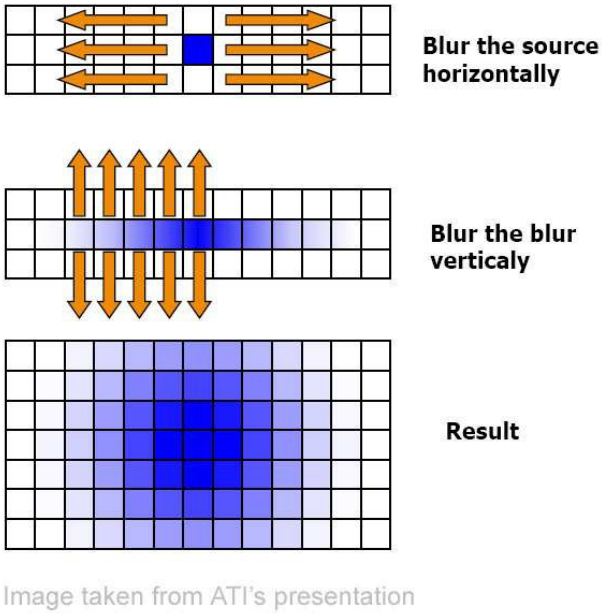

Figura 4.6: Separabilidade de filtros gaussianos

resoluções filtradas pela mesma gaussiana. É fácil perceber que os resultados de 4.7 e 4.8 são praticamente idênticos para um observador.

Existem diferentes filtros que podem ser aplicados para modelar diferentes efeitos de iluminação baseados no mesmo conceito base. [13] é uma boa referência para tais efeitos.

4.2.5 Combinação

Após a geração da imagem borrada, esta é combinada com a imagem original. Nesse estágio, as imagens ainda estão em formato de alta precisão e ainda não podem ser mostradas no monitor. As imagens processadas sofrem um processo de reescala para seu tamanho original. Nesse processo, esta será distorcida e terá uma aparência mais borrada. Como os efeitos de iluminação associados à HDR se beneficiam desse aspecto, tal distorção não acarreta problemas.

4.2.6 Tonemapping

O estágio de *tonemapping* mapeia valores valores HDR em valores LDR que podem ser exibidos numa superfície tradicional de renderização e consequentemente visualizados no monitor. Existem diferentes algoritmos de tonemapping. Os mais tradicionais algoritmos de tone mapping podem ser

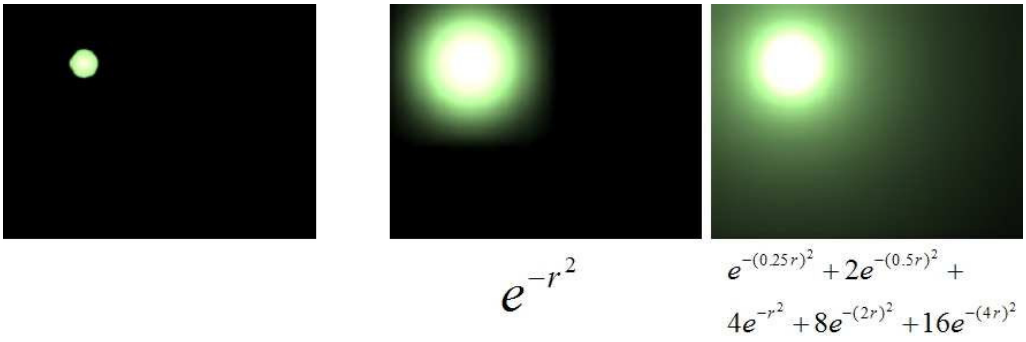

Figura 4.7: Combinação de gaussianas com diferentes raios aplicadas em uma mesma imagem. A imagem original é apresentada a esquerda, as imagens a direita são o resultado da aplicação do filtro gaussiano de raio unitário e da combinação de diferentes raios.

encontrados em [47], e sua escolha depende, basicamente das características do resultado final desejado, seja precisão de representação, questões de aspecto artístico, etc. Nessa fase é aplicada um dos mais interessantes efeitos HDR, o controle de exposição de imagens, que permite modificar toda a iluminação da cena, sem modificar qualquer parâmetro das luzes utilizadas durante a renderização da cena. Como os dispositivos de visualização atuais não possuem capacidade para exibir as superfícies HDR [47], é necessário um conjunto de operações que permitam tal visualização. Como dito no Capítulo 1, uma das áreas mais importantes da pesquisa em imagens HDR está no mapeamento dos valores de radiância codificados nessas imagens, em valores de brilho de pixel, suportados pelos dispositivos de exibição atuais.

Sem um bom algoritmo de tonemapping, a qualidade do resultado visual que a renderização em alta precisão permitiria, pode simplesmente ser perdido, no pior caso, ocorrendo a saturação dos valores não suportados pelo dispositivo.

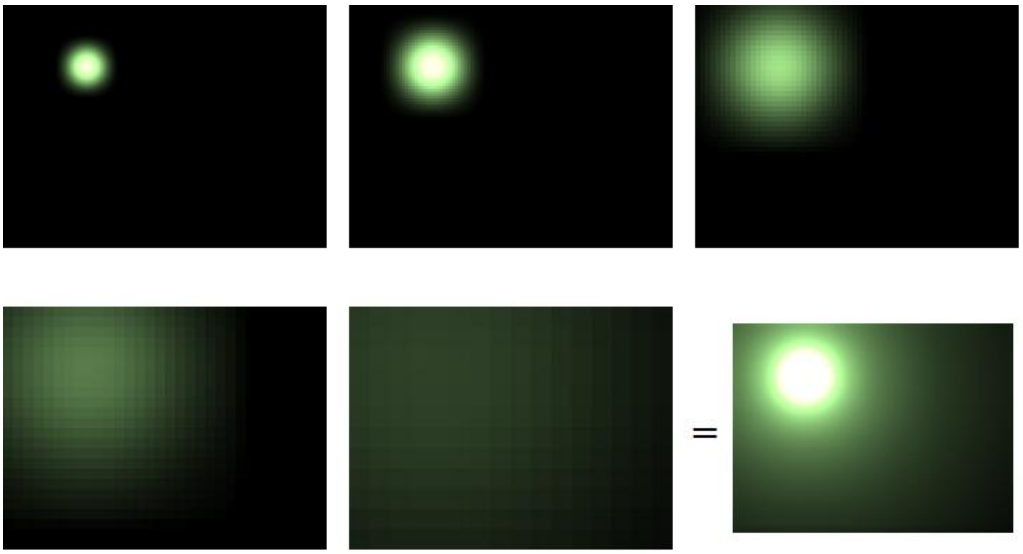

Figura 4.8: Imagens filtradas de diferentes tamanhos combinadas para gerar o resultado final