Universidade Federal de Santa Maria
Departamento de Computação Aplicada
Prof. Cesar Tadeu Pozzer
Disciplina: Computação Gráfica
pozzer@inf.ufsm.br
17/08/2017

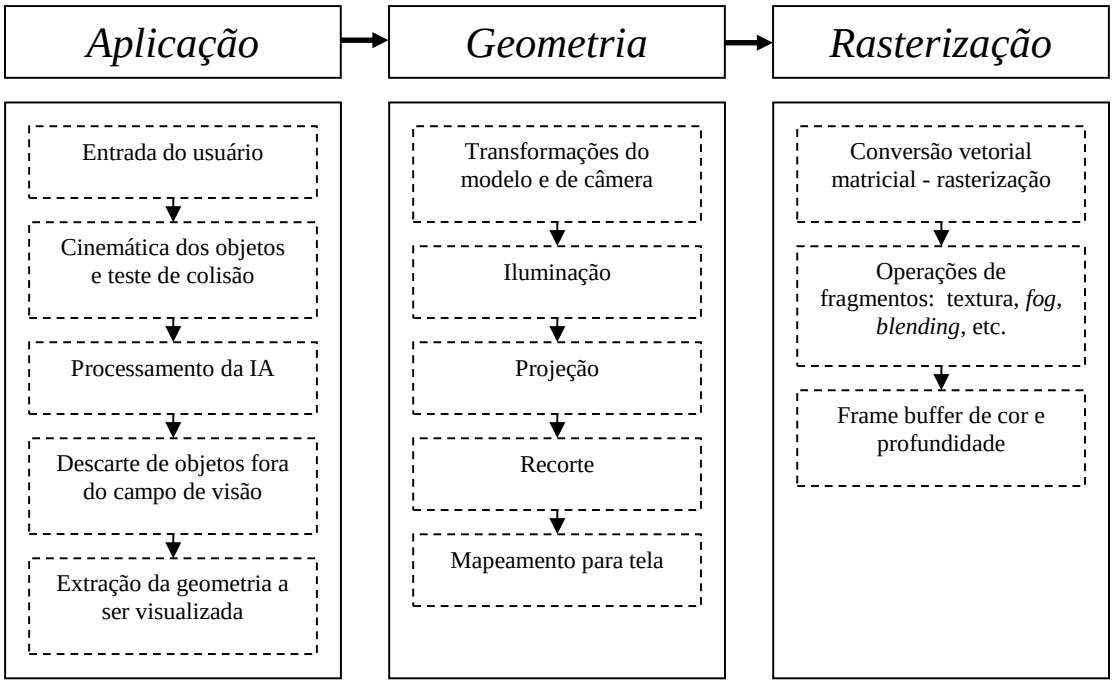
Pipeline de Renderização – OpenGL 1.X

Uma corrente não é mais forte que o seu elo mais fraco. – Anônimo

O pipeline gráfico de uma aplicação gráfica que faz uso do OpenGL (sem linguagem de Shader) é composto por 3 etapas: aplicação, geometria e rasterização. Este pipeline mudou com a inserção do pipeline programável, que será visto na sequencia da disciplina.

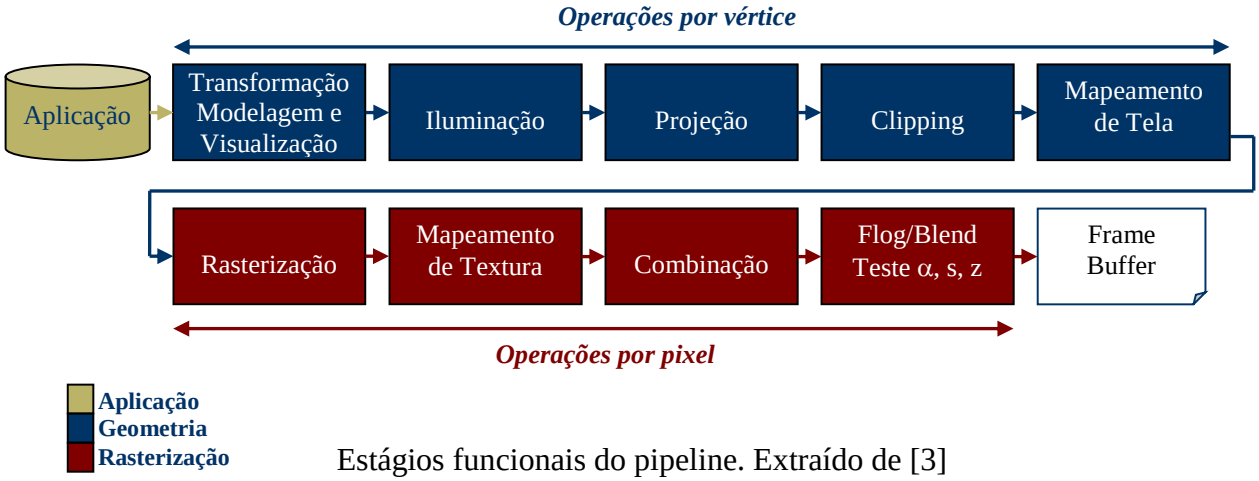
Os processos do nível de aplicação executam na CPU e os dois demais na GPU. **Os 3 estágios executam simultaneamente**. Antes do surgimento de placas aceleradoras, todo este trabalho era executado exclusivamente pela CPU. Hoje em dia há uma separação das tarefas, o que permite aliviar o uso da CPU para outras tarefas como:

- processamento da IA (visto que ainda não existem placas destinadas a este propósito),
- interação com o usuário,
- simulação física (por pouco tempo, visto que estão surgindo placas dedicadas ou incorporadas à placa de vídeo destinadas a este propósito),
- Algoritmos de remoção de objetos não visíveis (*culling*), etc.



Etapas do processo de renderização. Adaptada de [3].

Como mostrado na seguinte figura, o pipeline de geometria opera sobre vértices, enquanto o pipeline de rasterização opera sobre pixels (fragmentos).



Estágios funcionais do pipeline. Extraído de [3]

O *Framebuffer* é composto por:

- ***Color buffers***: guarda a cor dos pixels (fragmentos). Pode existir o *left-* e *right-buffer* (para visão estéreo), e cada um destes pode ser ***single*** ou ***double*** (*front-* e *back-buffer*).
- ***Depth buffer***: guarda a profundidade dos fragmentos em relação à posição da câmera. Podem haver vários fragmentos que projetam em um mesmo pixel. Somente o mais próximo será o visível (ver algoritmo z-buffer).
- ***Accumulation Buffer***: semelhante ao buffer de cor, com a diferença que este pode ser usado para acumular uma série de imagens. Isso é útil em operações de super-amostragem, antialiasing e *motion blur*.
- ***Stencil Buffer***: buffer usado para restringir o desenho a certas partes da tela. Pode ser usado para desenhar o interior do carro (painel, direção, etc). Somente a área na região do pára-brisa é que pode ser atualizada com informações da cena.

Para cada buffer, existem comandos de limpeza, como `glClearColor()`, `glClearDepth()`, `glClearAccum()` e `glClearStencil()`, ou `glClear(GL_COLOR_BUFFER_BIT | ...)`. Para maiores detalhes de uso destes buffers, consulte [1].

O buffer de profundidade é alterado pelo **algoritmo z-buffer**. Este buffer tem o mesmo tamanho e forma que o color buffer, onde cada pixel armazena a distância da câmera até a primitiva corrente mais próxima.

- Quanto uma primitiva estiver sendo renderizada para um certo pixel (**amostragem**), compara-se a distância deste pixel em relação à câmera.
- Se a distância for menor que a distância já armazenada, significa que a primitiva corrente, para o dado pixel, está mais próxima que a última primitiva avaliada. Neste caso deve-se atualizar o buffer de profundidade, juntamente com o buffer de cor com informações da nova primitiva.
- Se a distância for maior, mantêm-se os buffers inalterados. O custo computacional do z-buffer é O(n), onde n é número de primitivas sendo renderizadas.

Otimização do Pipeline

Como os 3 estágios conceituais (aplicação, geometria e rasterização) executam simultaneamente, o que for mais lento (**gargalo**) vai determinar a velocidade de renderização (*throughput*).

Observações:

- Antes de tudo deve-se localizar o gargalo;
- Deve-se procurar otimizar o estágio mais lento para aumentar o desempenho do sistema;
- Se o estágio da aplicação for o mais lento, não se terá nenhum ganho otimizando o estágio de geometria, por exemplo;
- Ao se otimizar o estágio mais lento, pode ocorrer de outro estágio se tornar o novo gargalo;
- Caso o gargalo não puder ser otimizado, pode-se melhor aproveitar os demais estágios para se obter maior realismo.
- Com a inserção do pipeline programável (GL 3.0), e unificação de vertex processor e pixel processor em stream processors, muitas das regras abaixo deixam de existir.

A estratégia mais simples para se **detectar o gargalo** é realizar uma série de testes, onde cada teste afeta somente um único estágio. Se o tempo total for afetado, ou seja, reduzir o frame rate, então o gargalo foi encontrado. Para isso pode-se aumentar ou reduzir o processamento de um estágio.

- **Teste no estágio da Aplicação**: A estratégia mais simples neste caso é reduzir o processamento dos outros estágios, pela simples substituição dos comandos glVertex*() por glColor*(). Neste caso os estágios de geometria e rasterização não vão ter o que processar. Se o desempenho não aumentar, o gargalo está na aplicação. Outra estratégia é incluir um laço de repetição que realiza N cálculos matemáticos. Se o desempenho geral piorar, o gargalo era a aplicação.

- **Teste no estágio de geometria**: É o estágio mais difícil de se testar pois se a carga de trabalho neste estágio for alterada, será alterada também em outros estágios. Existem duas estratégias de teste: aumentar ou reduzir o número de fontes luminosas, o que não vai afetar os demais estágios, ou verificar se o gargalo está no estágio da aplicação ou rasterização.

- **Teste do estágio de Rasterização**: É o estágio mais fácil de ser testado. Isso pode ser feito simplesmente aumentando-se ou reduzindo-se o tamanho da janela onde as imagens estão sendo renderizadas. Isso não interfere nem na aplicação e nem na geometria. Quanto maior for a janela, maior será a quantidade de pixels a serem desenhados. Outro modo de testar é desabilitando operações de textura, *fog*, *blending*.

Existem várias estratégias para se otimizar os estágios do pipeline. Algumas abordagens comprometem a qualidade em troca de desempenho e outras são apenas melhores estratégias de implementação:

- **Otimização do Estágio da Aplicação**: otimizações para tornar o código e acesso à memória mais rápidos. Isso pode ser obtido ativando-se flags de compilação e conhecendo detalhes das linguagens de programação. Algumas dicas gerais de código/memória são listadas. Para maiores detalhes consulte [2]:
  - Evite divisões;
  - Evite chamar funções com pouco código. Use funções inline;
  - Reduzir uso de funções trigonométricas;
  - Usar float ao invés de double;
  - Usar const sempre que possível, para o compilador melhor otimizar o código;
  - Evitar uso excessivo de funções malloc/free.

- **Otimização do Estágio de geometria**: Pode-se aplicar otimizações apenas sobre transformações e iluminação.
    - Em relação a transformações, se o objeto precisar ser movido para uma posição estática, ao invés de aplicar a transformação a cada renderização, pode-se aplicar as transformações em um pré-processamento antes da renderização iniciar;
    - Outra estratégia é reduzir ao máximo o número de vértices, com o uso de vertex array, ou primitivas do tipo STRIP ou FAN;
    - Outra estratégia muito eficiente é aplicar culling, no estágio da aplicação;
    - Uso de LOD;
    - Reduzir o número de fontes luminosas;
    - Verificar quais partes da cena necessitam iluminação;
    - Iluminação flat é mais rápida que Gourdaud;
    - Avaliar se é necessário iluminar os dois lados das superfícies;
    - Uso de *Light maps*.

- **Otimização do Estágio de Rasterização**:
    - Habilitar *backface culling* para objetos fechados. Isso reduz o número de triângulos para serem rasterizados em aproximadamente 50%;
    - Desenhar objetos mais próximos do observador primeiro. Isso evita gravações sucessivas no z-buffer;
    - Desabilitar recursos como fog, blending ou uso de filtros de textura mais baratos, como o bilinear ao invés de mipmapping.

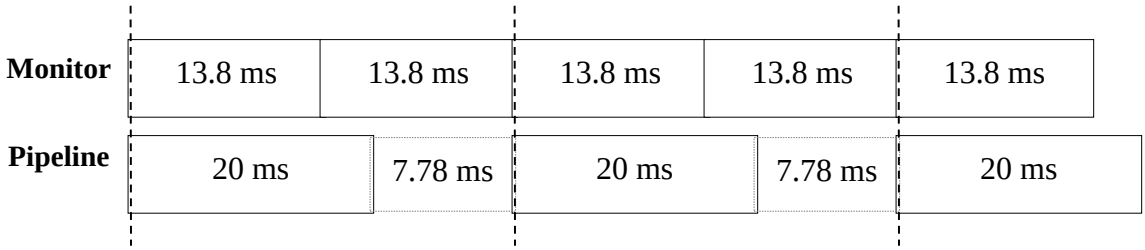
- **Otimizações gerais**:
    - Reduzir a geometria da cena;
    - Reduza a precisão em vértices, normais, cores e coordeandas de textura. Isso reduz uso de memória e uso de barramento;
    - Desabilitar recursos não utilizados;
    - Minimizar mudança de estados;
    - Evitar swap de texturas;
    - Use display lists para objetos estáticos.

11.2. Avaliando o Desempenho do Pipeline

A velocidade de renderização depende do estágio mais lento, e é medida em FPS (frames por segundo). Para este cálculo, deve-se desabilitar o recurso de *double-buffer*, visto que a troca de buffer neste caso ocorre em sincronismo com o refresh do monitor.

Com o double buffer ativado, enquanto o *front buffer* está sendo exibido, o *back buffer* está sendo gerado. Somente ocorre a troca de buffers (`glutSwapBuffers`) quando o monitor acabar de desenhar o frame corrente.

Considerando-se o monitor com uma taxa de atualização de 72 Hz, significa que a cada t = 13.8 milisegundos a tela é atualizada. Com o recurso de *double-buffer* ativado, a taxa de atualização deve ser múltiplo deste valor. Se o tempo de renderização consome 20 ms (50 Hz) em *single-buffer*, a taxa cairá para 2*13. 8 = 36 Hz com o *double-buffer* ativo. Neste caso, o pipeline gráfico fica inativo por 13. 8*2 – 20 = 7.78 ms para cada frame [2].



Uma forma de contornar os problemas do uso do *double-buffer* é o uso de *tiple-buffer*. Neste caso existem dois *back buffers* e um *front buffer*. Neste caso, enquanto o monitor está exibindo o *font buffer*, podem estar sendo gerados 1 ou 2 *back buffers*. No caso do uso de *double-buffer*, o o refresh do monitor for de 60 Hz e o tempo de geração das cenas for menor que 1/60 s, então mantém-se uma taxa de 60 fps. Porem se o tempo de geração for pouco maior que 1/60 s, a taxa de atualização cai para 30 fps. Com o uso de *triple-buffer*, a taxa se mantém próxima de 60 fps. O inconveniente do uso de *triple-buffer* é que o tempo de resposta para ações do usuário é um pouco maior, especialmente se o fps for baixo.