Introdução a Jogos Digitais

Aula 10 - Motores de Jogos II

Apresentação

Olá, pessoal!

Chegamos à nossa última aula desta disciplina. Aqui detalharemos um pouco mais sobre os principais componentes dos motores de jogos e, assim, teremos uma ideia do que eles podem fazer por vocês! Bom, já sabemos o quanto ajudam, agora saberemos de que formas!

Aposto que depois desta aula vocês ficarão ansiosos para conhecer as próximas disciplinas e começar a mexer em um motor de jogos!

Vamos lá!

Objetivos

- Entender de que forma os motores de jogos e seus componentes auxiliam no desenvolvimento de jogos;
- Conhecer os principais componentes de um motor de jogos:
    - Motor gráfico
    - Motor de física
    - Motor de áudio

1 - Liguem os Motores!

Na aula passada, vimos que a programação de jogos exige uma série de cuidados e que o motor facilita muito a nossa vida. Também vimos várias formas de como ele faz isso, abstraindo as funcionalidades mais difíceis de programar e deixando o desenvolvedor focado na lógica do jogo. Assim, os motores são ótimas invenções e resolvem todos os nossos problemas! Ou pelo menos aqueles que nos dariam dores de cabeça na hora de programar os nossos jogos!

No entanto, as coisas não são tão simples assim! Acho que será melhor se entendermos um pouquinho mais como tudo começou.

Figura 01 - Senta que lá vem a história!

Fonte: http://www.casadobrincar.com.br/site/videos-infantis-senta-que-la-vem-a-historia/. Acesso em: 05 set. 2016.

Lembra que na aula passada nós falamos que algumas empresas de jogos começaram a desenvolver ferramentas que implementavam partes do jogo que se repetiam em diferentes projetos? Antigamente, o código era misturado entre a parte da lógica do jogo e o código necessário para a aplicação rodar e usar de forma

adequada e eficiente o hardware disponível. Isso dificultava muito para que esse código pudesse ser reutilizado, pois você precisava identificar dentro de um código-fonte gigante quais partes tratavam de cada coisa. Uma bagunça!

E então veio DOOM. Um jogo que não só alavancou o gênero do FPS (e ajudou a definir vários padrões para esse gênero), mas também contribuiu significativamente para o desenvolvimento de jogos.

Figura 02 - DOOM, um marco no desenvolvimento de jogos.

Fonte: http://www.pocket-lint.com/news/137607-doom-is-back-how-has-it-changed-over-23-years. Acesso em: 05 set. 2016.

A equipe que desenvolveu DOOM se preocupou em projetar o jogo com uma separação dos componentes de programação e dos recursos de arte. Dessa forma, uma equipe de artistas poderia criar novos recursos e fazer um “novo” jogo (com a parte de funcionamento igual ao DOOM) no que concerne ao visual. Esse primeiro esforço de separação entre a parte da programação e a parte artística fez com que o termo “motor” fosse utilizado pela primeira vez para se referir a essa aplicação base que poderia ser alterada para criar novos jogos.

Perceba que essa arquitetura não era tão flexível quanto a que temos hoje em dia, mas apresenta uma noção muito importante do funcionamento dos motores: a **modularização**. Os componentes são separados de acordo com a sua funcionalidade, assim, partes que tratam de uma mesma atividade (desenhar coisas na tela, por exemplo) estão juntas na ferramenta. Isso facilita dois aspectos do processo de desenvolvimento:

- **Facilidade para isolar erros:** se o jogo começa a apresentar personagens com falhas no desenho, pode-se testar os diferentes sistemas envolvidos (desenho em tela, carga do modelo 3D) e detectar em que ponto o erro está acontecendo. Em um código onde as funcionalidades estão misturadas e entrelaçadas, é mais difícil detectar onde a falha ocorre. Com a modularização por funcionalidades, é possível testar apenas a parte do desenho e, não detectado o erro, pode-se testar a função que carrega os dados do modelo 3D na memória. Menos dores de cabeça para os programadores, com certeza!

- **Segurança para fazer alterações:** se for necessário mudar um algoritmo que renderiza a iluminação pontual (simula a iluminação de uma lâmpada, por exemplo), é mais fácil trabalhar em um código no qual você sabe que todas as instruções interferem apenas no desenho da luz do que em um código no qual você pode acidentalmente alterar a lógica do jogo!

Figura 03 - A felicidade dos programadores quando usam um motor de jogos!

Fonte: http://knowyourmeme.com/memes/first-day-on-the-internet-kid. Acesso em: 05 set. 2016.

Perceba que a separação proposta por DOOM ainda é bastante limitada: usando o motor do jogo, você só conseguia criar jogos parecidos com DOOM, mudando a arte. Com o tempo, novas propostas de modularização foram surgindo, gerando vários níveis de flexibilidade para o motor. Primeiramente, isso se restringiu a motores para jogos do mesmo gênero, ou seja, motores que possuíam otimizações para problemas comuns em jogos FPS ou de estratégia, por exemplo. Em *Counter Strike*, um jogo de FPS, o modo como a câmera e o nível de detalhe em que o personagem do jogador e os adversários são exibidos na tela é totalmente diferente de um jogo de estratégia em tempo real como *Starcraft*. Esses motores buscavam facilitar o desenvolvimento trazendo as melhores implementações de funcionalidades comuns (iluminação, renderização do espaço, representação dos elementos do jogo em memória, câmera do jogo, geração de rotas dos personagens no jogo, etc.). Com o tempo, motores mais generalistas passaram a ser criados, de forma que o mesmo motor pudesse ser usado para construir jogos em qualquer

gênero, como é feito no Unreal 4 e no Unity. A diferença entre essas abordagens ocorre no grau de reuso que elas permitem. Vamos pensar em um exemplo para clarear essa questão!

Imagine que você deseja fazer um jogo de corrida. Uma das funcionalidades que, com certeza, precisará é desenhar o carro na tela, não é mesmo? Um motor poderia ter diferentes abordagens para isso:

- Um motor específico para jogo de Fórmula 1 poderia ter uma função *desenhaCarro*, que só conseguiria desenhar carros no modelo de um carro de circuito de Fórmula 1, permitindo ao desenvolvedor apenas a escolha das cores.

- Um motor para jogos de corrida poderia ter uma função *desenhaCarro*, na qual o desenvolvedor informaria o tipo do carro a ser desenhado (Fórmula 1, Ferrari, Fusca) e o motor saberia como desenhar baseado no modelo.

- O motor poderia ter uma função *desenhaModelo*, capaz de desenhar qualquer objeto a partir de uma malha 3D fornecida pelo desenvolvedor. Além de desenhar os carros, ele poderia desenhar pessoas, animais, etc.

Existe uma relação de custo/benefício entre flexibilidade de uso e facilidade de desenvolvimento. É óbvio que, se o seu foco é apenas um jogo de Fórmula 1, o primeiro motor já seria o suficiente. Você não precisaria se preocupar em desenhar o modelo do carro, pois ele já estaria pronto! No entanto, você ficaria limitado a trabalhar apenas com os elementos que ele lhe fornecesse, então, se quisesse colocar aquela paisagem linda com montanhas rochosas e o motor não deixasse, poderia esquecer. No outro extremo, temos o motor que deixará você fazer o que quiser, porém precisará efetivamente fazer várias coisas, pois ele lhe dá uma funcionalidade básica de desenho em tela a partir de um modelo 3D, mas você terá de produzir todos os modelos para ele.

Figura 04 - Quanto mais flexível o motor, mais você pode dar o seu toque pessoal para o jogo.

Fonte: http://www.readingtree.org/top-10-car-racing-games-to-play-in-2013/. Acesso em: 05 set. 2016.

E já que falamos em reuso, deixe-me aproveitar o exemplo para falar de outro ponto! Imagine um motor com a função *desenhaCarro* que só consegue desenhar carros de Fórmula 1. De repente, ela resolve modificar essa função para que os desenvolvedores de jogos possam usar a ferramenta para desenhar não apenas um tipo de carro, mas também outros veículos. Uma grande vantagem de uma ferramenta modularizada diz respeito à definição das interfaces de como usar as funcionalidades, permitindo a transparência sobre como elas são implementadas. Agora imagine que, para usar a função *desenhaCarro*, o desenvolvedor do jogo só precise escrever a seguinte linha:

desenhaCarro(numero1)

Onde *numero1* é o carro que ele quer desenhar. Ao alterar a função, a empresa que desenvolve o motor pode mudar a interface da função e dizer que a partir de agora o jogo precisa informar o carro e o tipo do modelo a desenhar, por exemplo:

desenhaCarro(numero1, "Ferrari")

OK, isso já é legal! Mas ainda não é perfeito, porque o desenvolvedor do jogo precisará fazer ajustes no seu código para se adequar à nova interface ou, então, usar uma versão antiga da ferramenta. Agora imagine uma função mais genérica,

como *iluminaSala*, que é responsável por pegar uma fonte de iluminação no mundo do jogo e simular todos os raios de luz da cena. Digamos que ela é usada assim:

iluminaSala()

Depois de muita pesquisa, a equipe desenvolvedora do motor descobre um jeito muito mais eficiente de fazer essa simulação, que dobrará a velocidade da renderização da luz e deixará os jogos mais rápidos consumindo menos memória. Mas, para isso, eles precisam implementar a função *iluminaSala* completamente, mudando inclusive as estruturas que são utilizadas internamente. A função que tinha 30 linhas passa a ter 500, porém a equipe não muda a interface de chamada.

Sabe o que o desenvolvedor de jogos usuário do motor precisa fazer no seu código para usar essa nova função? Nada. Continua chamando-a *iluminaSala()*. E essa é uma grande vantagem que se obtém com a **transparência** de funcionalidades do motor: você não precisa saber como a coisa funciona internamente, precisa apenas saber como utilizá-la.

Essa discussão toda foi para falar que nenhum motor é perfeito, mas o uso de um deles torna a sua tarefa muito mais fácil tanto do ponto de vista técnico como no ponto de vista de projeto. Porém, mesmo com uma gama diferente de motores, existem alguns componentes que são comuns entre eles, por serem necessários em praticamente todos os jogos. Falaremos um pouco sobre cada um deles!

2 - Principais Componentes de um Motor de Jogos

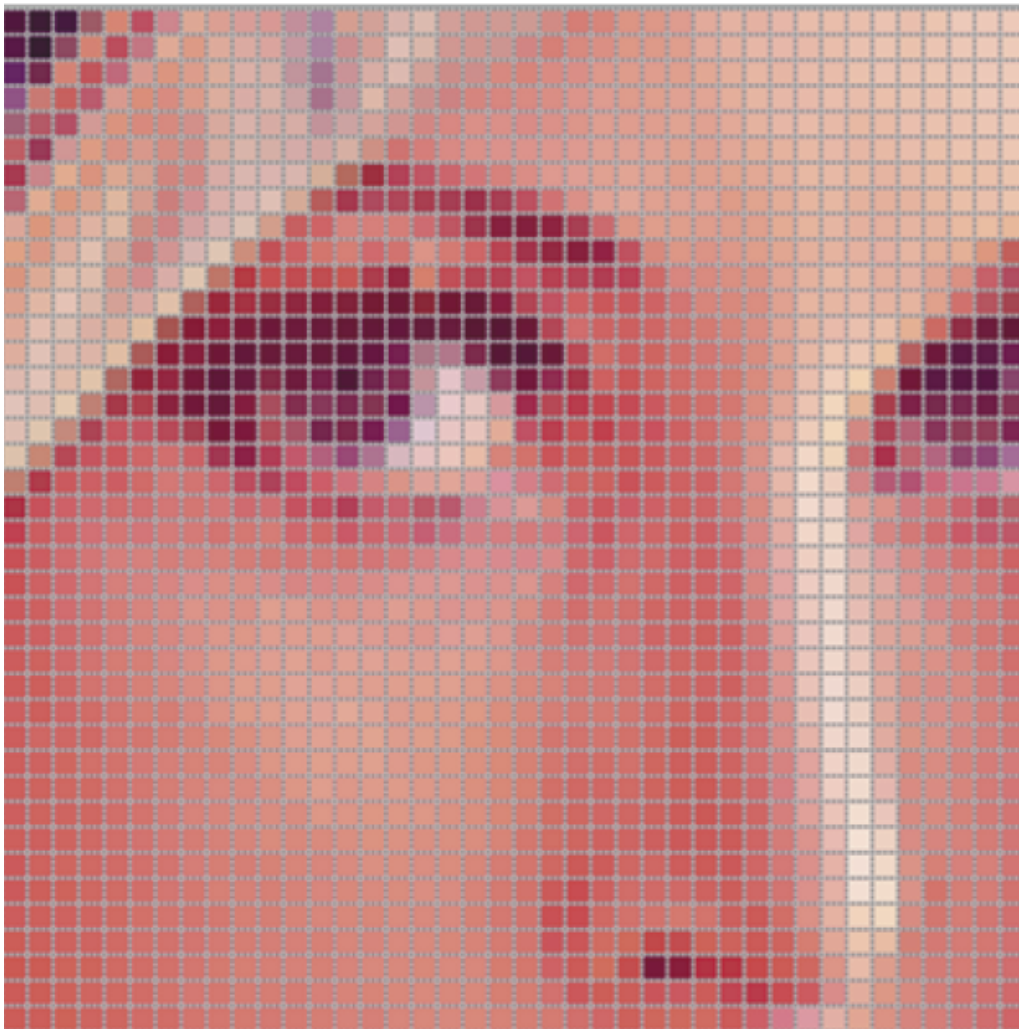
2.1 - Gráficos

A parte do motor gráfico é responsável por renderizar, ou seja, desenhar na tela os objetos do jogo. Você sabe como esse processo ocorre? Tentarei apresentar uma versão simplificada, mas certamente seria um assunto que daria várias aulas! Nas disciplinas sobre Motores de Jogos e Modelagem, vocês terão a oportunidade de entender um pouco mais sobre as questões gráficas no desenvolvimento de jogos.

Para desenhar a imagem do jogo no nosso monitor, é necessária uma sequência de passos, que levam desde os arquivos binários que representam os objetos 3D até o conjunto de pontos coloridos que formam as imagens no nosso monitor. Cada

pontinho desses é chamado de **pixel** e representa a menor parte que conseguimos manipular no nosso monitor. Sabe quando falamos de resolução de tela? 1.920 por 1.080? Estamos dizendo que o nosso monitor mostrará a imagem com 1.920 pixels na horizontal e 1.080 na vertical (essa é a resolução full HD, por sinal). Nossa imagem é formada quando juntamos esses vários pontinhos.

Figura 05 - As imagens exibidas no nosso monitor são formadas por vários pixels, cada um com sua cor.



Fonte: https://jackkyriacou.wordpress.com/2007/12/18/pixels/. Acesso em: 05 set. 2016.

Ok, professor, mas por que você está falando disso? É porque nos ajudará a entender vários passos desse processo de renderizar as imagens na tela, normalmente chamado de **pipeline gráfico**. De uma forma geral, os passos são:

- Ler os dados representados em um arquivo;
- Gerar os modelos 3D;

- Posicionar a câmera do jogo;
- Fazer uma projeção do mundo 3D para a tela 2D;
- Realizar algumas (muitas) otimizações;
- Rasterizar a imagem para a tela do monitor, entre outros ajustes.

Então, continuando... O quê? Vocês querem saber mais sobre cada passo? Está bem, veremos um pouquinho mais a respeito deles. Você já parou para pensar como uma imagem é representada no computador?

Figura 06 - Uma imagem 2D do encanador mais querido do mundo dos games!

Fonte: http://www.giantbomb.com/2d/3015-1427/. Acesso em: 05 set. 2016.

Se você der um zoom, conseguirá ver os pixels da imagem. No seu computador, provavelmente a imagem estará salva em um arquivo (JPG, GIF, PNG) no qual há um conjunto de informações dizendo quantos pontinhos iremos pintar e as suas respectivas cores. Suponha que o arquivo representante da imagem é composto por vários números da seguinte forma:

100 200 125 250 255 120 78 88 157...

Os sistemas de cores que temos no computador normalmente utilizam uma combinação de três valores para determinar uma cor. Um exemplo disso é o RGB, um sistema famoso que utiliza as cores vermelha (***R**ed*), verde (***G**reen*) e azul (***B**lue*) de forma misturada para compor um esquema complexo de cores (sacaram o porquê do nome RGB?). Dessa forma, cada número do arquivo representa uma cor que deve ser desenhada na tela: 100 200 125 representa as informações do primeiro ponto colorido e assim por diante. Mas apenas ter esse arquivo não é o suficiente! O computador precisa saber ler essas informações e transformar isso em uma imagem. Sabe quando você pega uma imagem, tenta abrir em um programa e ele diz que o formato é desconhecido, ou dá um erro e não consegue abrir? O que falta para o programa são as instruções de como ler os dados daquela imagem, pois ele precisa saber que a cada três números deve desenhar um pixel de acordo com os valores especificados.

Da mesma forma que uma imagem é definida por um arquivo de dados, os modelos 3D usados na criação de jogos também são! E com mais informações: cada ponto do modelo 3D tem valores de posição especificados nos três eixos de coordenada, e esse conjunto de pontos (chamados de malha) define a forma que o modelo terá.

Figura 07 - Um exemplo de modelo 3D. Eita que bicho feio!

Fonte: http://gamedev.stackexchange.com/questions/33016/how-does-3d-games-work-so-fluent-provided-that-each-meshs-size-is-so-big. Acesso em: 05 set. 2016.

Existem diferentes formas de construir esse modelo através do conjunto de pontos: algumas são mais complexas e utilizam equações matemáticas que delineiam com precisão a forma do objeto. Outras são mais eficientes, quebrando o objeto em formas menores, porém mais simples de se desenhar, como triângulos e quadrados. Em jogos, a segunda opção é mais comum, já que os gráficos são gerados em tempo real e precisam de alto desempenho. A primeira forma é mais comum na produção de animações, em que cada cena leva dias para ser renderizada em computadores potentes, processando com exatidão cada parte dos objetos que ganharão vida no filme.

Figura 08 - Em animações de alta qualidade, é comum utilizar várias máquinas potentes que passam horas para renderizar um quadro do filme!

Fonte: http://pt-br.finalfantasy.wikia.com/wiki/Personagens_de_Final_Fantasy_VII:. Acesso em: 06 set. 2016.

Bom, gerado o modelo 3D dos objetos, acabaram os problemas? Não, ainda tem muito chão pela frente. Para continuar o processo de renderização, o próximo passo é definir qual ponto de vista será mostrado desse mundo, isto é, a janela de visualização, sendo esta o local onde estará a câmera do jogo. Essa câmera pode ser colocada para nos passar a visão do personagem, como nos jogos FPS, ou ser mais aberta e nos dar uma visão mais ampla do mundo do jogo, como nos jogos de

estratégia ou de aventura. Duas coisas são importantes na definição da posição da câmera: saber quais objetos aparecerão naquele ângulo de visão e a que distância eles estão.

Definida a câmera, vamos para um passo matemático importantíssimo: transformar a nossa visão do mundo 3D em uma projeção 2D. Afinal, nosso monitor é 2D (ainda). Para isso, aplicamos várias contas que determinam como os elementos serão exibidos de acordo com a posição da câmera, inclusive os efeitos de como a iluminação do jogo afeta a visualização de cada objeto na cena. As distâncias e os tamanhos de cada objeto, além das possíveis distorções que ocorreriam na conversão de uma visão 2D de um mundo 3D, são ajustados nesse momento, para que não se perca a relação espacial entre os vários objetos no mundo do jogo.

Feito isso, o próximo passo é a realização de diversas operações de otimização. Essas otimizações vão desde eliminar objetos que não estão dentro da área visível da câmera, até mesmo remover partes de objetos que estão “escondidas” por outros objetos, entre outras ações as quais reduzem a quantidade de elementos que deverão ser tratados na geração da imagem final. Tudo pelo desempenho!

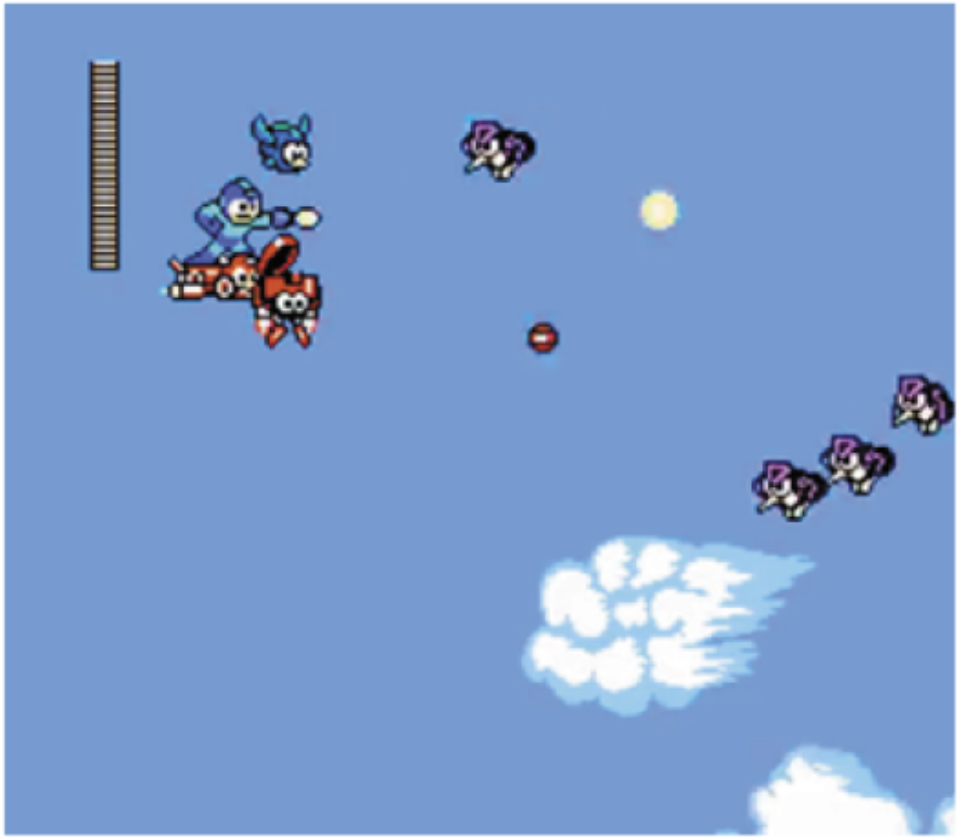
Um dos passos finais é a **rasterização**: o que não foi cortado na otimização agora é convertido em uma imagem que será exibida para o jogador na tela de sua plataforma. Essa conversão é bem complexa, mas basicamente o que ocorre é uma sequência de operações que determinam as propriedades de cada pixel que irá compor essa imagem final.

Agora pressuponha que você precisasse fazer isso manualmente todas as vezes que quisesse criar um jogo! Ainda bem que existem os motores e componentes gráficos, não é mesmo? Outro detalhe: os jogos 2D, apesar de serem mais simples e não possuírem o mesmo nível de complexidade em todas as etapas, ainda passam por vários passos desse pipeline até a renderização das informações na tela.

2.2 - Física

O motor de física é o componente responsável por realizar as simulações de como as forças físicas e os movimentos dos objetos do jogo ocorrem. Imagine aquela fase do Megaman, com 10 inimigos vindo na sua direção, balas para todos os lados, fazendo você dar pulos duplos pelas paredes...

Figura 09 - Megaman voando nas costas de um cachorro robô, enquanto luta contra pássaros cibernéticos. Because physics, right?



Fonte: http://www.coronajumper.com/2016/05/rockman-8-fc.html. Acesso em: 06 set. 2016.

Pulos duplos? Pois é, a simulação de física dos jogos não precisa ser necessariamente realista, apenas coerente para que o jogador entenda como os elementos do jogo se comportam. Até porque uma simulação física realista envolve tantas variáveis que seria extremamente complexo (e custoso) calcular um cenário de um jogo, ocasionando uma enorme perda de desempenho. E a regra nos jogos é: tudo para ganhar desempenho! Uma facilidade que o motor de física fornece para o desenvolvedor é que ele não precisa se preocupar em codificar rotinas para cada elemento do jogo, visto que o motor possui algoritmos padrões para simular os eventos e aplicá-los para todos os objetos de uma fase ou cenário.

Uma das principais funções da parte física é a detecção de colisões entre os elementos do jogo. Será que o golpe/tiro bateu no adversário? Será que ele cai na plataforma ou escorrega pela ponta? Para resolver essas situações, é necessário que o jogo consiga entender se dois objetos colidiram, ou seja, se eles se tocaram. Existem diferentes técnicas para se detectar colisões nos jogos: o ***Bounding Box***

desenha uma caixa em volta do objeto do jogo, e se as caixas de dois objetos ocuparem o mesmo lugar no espaço, houve uma colisão. A vantagem desse método é a simplicidade em testar se houve uma colisão ou não (testa se os quadrados interseccionam), porém, ele não é muito preciso, além de ser comum vermos problemas como “colisões fantasmas” (os objetos batem sem se tocar!). Um método mais refinado é o ***Convex Hull***, ou envoltório convexo: nele, a área de colisão segue a forma do objeto, como se tivéssemos embrulhado-o num papel de presente. Dessa forma, a colisão pode ser feita em relação a partes do objeto, como os braços do boneco ou a sua cabeça. Apesar de ser mais complexo, esse método permite a detecção de colisões de forma mais precisa, inclusive criando efeitos diferenciados dependendo do ponto da colisão.

Figura 10 - Está vendo a figura acima? Na imagem à esquerda temos um exemplo de como funciona o Bound Box, enquanto que na imagem à direita temos um exemplo de como funciona o Convex Hull.

Uma otimização muito importante que o motor faz é a de “congelar” objetos que não estão dentro do raio de interação do jogador. Imagine um mapa imenso, cheio de adversários, como os do jogo *Far Cry*, por exemplo: se a cada instante o jogo precisasse calcular a física da movimentação de todos esses objetos, era capaz de até mesmo o mais potente computador “suar” para rodar o jogo! Para que isso não ocorra, o motor de física realiza a simulação apenas dos elementos que são de interesse para o jogador, enquanto outros que não estão visíveis ou próximos a ele ficam em um estado adormecido, não consumindo processamento da plataforma. Eu já falei que nos jogos era tudo pelo desempenho? :)

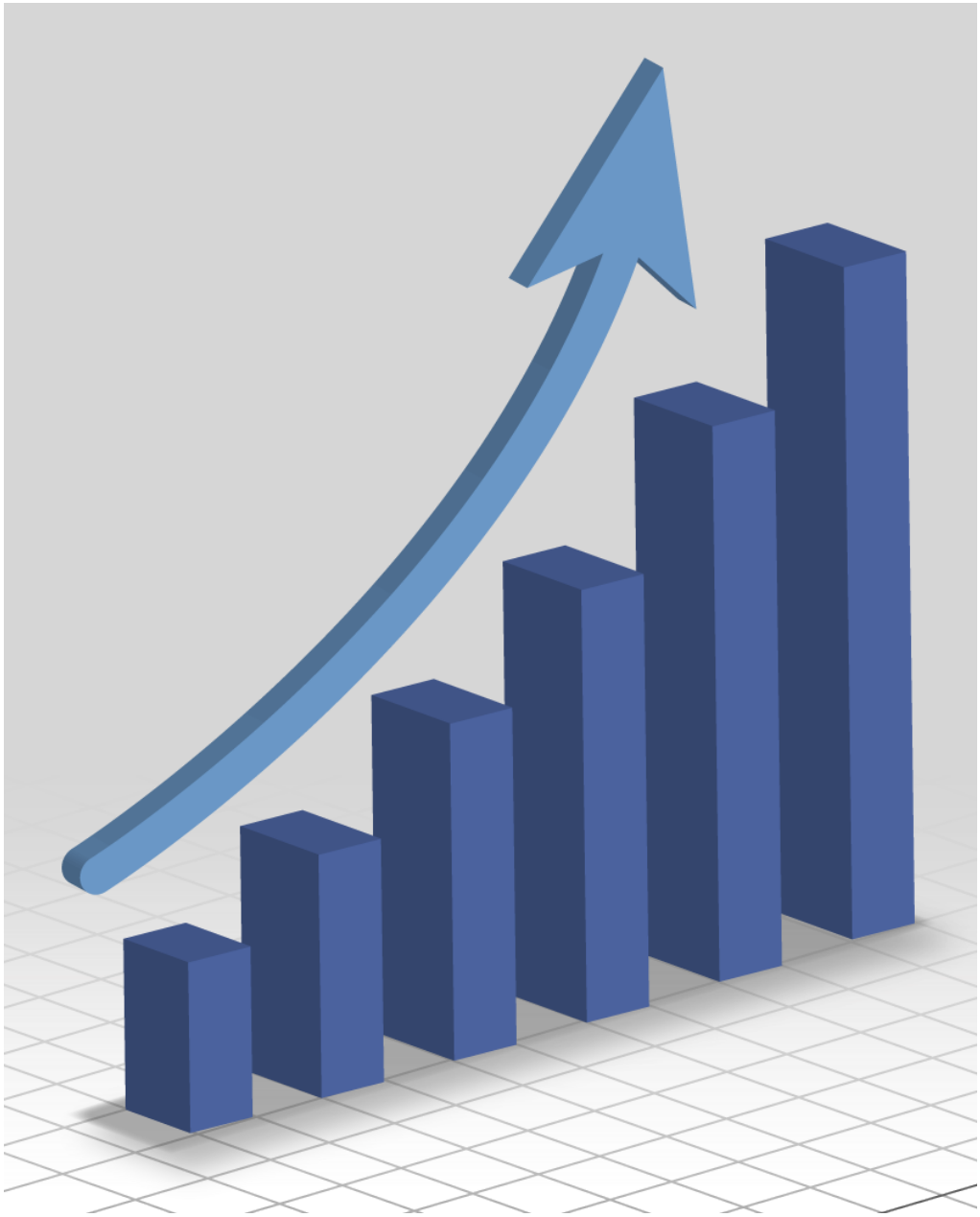
2.3 - Áudio

O áudio é um componente extremamente importante de um jogo. Já jogou um jogo no mudo? Não é a mesma coisa! Agora qual seria a dificuldade de colocar o som para tocar no jogo?

Vamos voltar às aulas de Arquitetura de Computadores (módulo básico, nem faz tanto tempo assim!): lembra de hierarquia de memória? Como era mesmo? Ah, existia uma diferença na velocidade de acesso entre memórias distintas! O HD, que tem bastante espaço para guardar arquivos, também tem um acesso muito lento, enquanto a memória RAM consegue uma velocidade muito melhor! Sem contar na diferença do preço entre elas!

Mas por que você está relembrando isso, professor? Qual a regra do jogo mesmo?

Figura 11 - DESEMPENHO!!!!!!!



Pois bem, para que não afete o desempenho do jogo, o áudio precisa estar armazenado em uma memória com velocidade de acesso mais rápida. Logo, o motor de áudio precisa estar constantemente garantindo que os arquivos com os sons os quais serão tocados estejam na memória RAM no momento que ele for necessário. Alguns arquivos podem ser tocados a qualquer momento, então eles já são carregados diretamente lá. Já outros são tocados em momentos específicos, como em uma conversa entre personagens, um evento no meio da fase, o som do golpe de um chefão. Esses sons precisam ser carregados momentos antes de serem tocados! Essa é uma tarefa complexa, que fica a cargo do componente de áudio do motor do jogo.

Figura 12 - As músicas da ocarina do Link no jogo Zelda: Ocarina of Time são carregadas em momentos específicos do jogo, quando o jogador abre o menu para tocar uma música.

Fonte: http://zelda.wikia.com/wiki/Epona's_Song. Acesso em: 06 set. 2016.

Além disso, o componente que cuida do áudio também faz várias outras tarefas, como suportar diversos formatos de arquivos e conseguir colocá-los para tocar sem que ocorra interrupções na música a cada ciclo de processamento do jogo. A sincronização da música e dos sons com os eventos na tela também é muito importante, já que o jogador pode perder a concentração no jogo se perceber que ocorre uma demora entre a ação e o som correspondente a ela. Igual a filme mal dublado!

Existem outros componentes que poderíamos abordar ainda nesta aula, como a parte de redes, animações e inteligência artificial. Poderíamos passar um módulo inteiro estudando sobre motores de jogos!

Mas vamos parar por aqui! O objetivo era apenas introduzir alguns dos principais componentes, e vocês terão mais disciplinas e oportunidades de aprender detalhes dos vários aspectos dos motores de jogos!

Até a próxima!

Pontos-Chave

Última aula da parte introdutória, que legal! Para não perder o costume, seguem alguns pontos de interesse desta aula:

- Uma das principais vantagens no uso dos motores de jogos é a modularização por funções que ele provê ao desenvolvedor de jogos.
- A transparência de implementação é uma característica que impacta diretamente na facilidade de se alterar as funcionalidades do motor sem prejudicar os desenvolvedores que o utilizam.
- O motor gráfico tem como principal função renderizar a tela do jogo, executando as tarefas e passos do pipeline gráfico.
- Uma das principais exigências dos jogos é que boa parte dos gráficos são gerados em tempo real, sendo necessário o uso de técnicas que priorizam o desempenho de processamento.
- O motor de física simula os aspectos de movimento e interação entre os objetos do jogo, tendo como destaque o tratamento de colisões e a simulação de forças atuantes sobre os objetos do jogo.
- O motor de áudio realiza tarefas essenciais à execução das músicas e efeitos sonoros do jogo, de forma a não impactar no desempenho do jogo, e sem interrupções que tirem a concentração do jogador.

Leitura Complementar

Para a leitura complementar desta última aula, deixarei um desafio!

<http://www.grandmaster.nu/blog/?page_id=118>

Esse link possui um tutorial mostrando como se faz um motor de jogos simples. Ele explica muito bem a parte de programação, então acho que vocês vão adorar!

Autoavaliação

1. Qual a função principal do componente gráfico de um motor?
2. Quais os passos necessários para gerar a imagem do jogo na tela da plataforma do jogador?
3. Quais as operações que o motor de física faz para o desenvolvedor de jogos?

Referências

GREGORY, Jason. **Game engine architecture.** CRC Press, 2009.

MCSHAFFRY, Mike. **Game coding complete.** Cengage Learning, 2012

WIKIPEDIA. Rendering (computer graphics. Disponível em <https://en.wikipedia.org/wiki/Rendering>. Acesso em: 30 ago. 2016.

WIKIPEDIA. Physics engine. Disponível em <https://en.wikipedia.org/wiki/Physics_engine>. Acesso em: 30 agosto 2016.