Desenvolvimento de Aplicações Gráficas Interativas com a Unreal Engine 4

Daniel Valente de Macedo [1]
Yvens Rebouças Serpa [1]
Maria Andréia Formico Rodrigues [1]

Data de submissão: 11.06.2015
Data de aceitação: 26.10.2015

Resumo: Neste trabalho, apresentamos os conceitos básicos para o desenvolvimento de aplicações gráficas interativas em tempo real usando a *Unreal Engine 4*, a qual teve recentemente o seu código-fonte disponibilizado pela *Epic Games*. Inicialmente, descrevemos a arquitetura da ferramenta e as principais classes que devem ser implementadas para o desenvolvimento de projetos usando a *Unreal*, tanto teoricamente, quanto através de exemplos práticos. Adicionalmente, detalhamos o *Blueprint Visual Scripting System*, um sistema de *script* visual, o qual funciona a partir de uma mecânica baseada em nós de controle para o desenvolvimentos de elementos interativos (regras de jogo, jogadores, câmeras, entrada de dados e *assets*). Primeiramente, mostramos todos os passos para o controle do comportamento de um personagem em terceira pessoa (TPS), assim como para a construção dos elementos e materiais do cenário. Em seguida, descrevemos o procedimento para a criação de mecânicas de interação entre o usuário e os diferentes objetos e elementos do cenário, incluindo o uso do editor de partículas, chamado *Cascade Particle Editor*. Finalmente, a partir da *Unreal Motion Graphics (UMG)*, uma ferramenta disponível na *engine*, geramos uma interface gráfica para a apresentação de dados e informações básicas do jogo (vida, munição, etc.).

Abstract: In this work, we present the basic concepts for developing interactive graphics applications in real time using the *Unreal Engine*, which recently had its source code available from *Epic Games*. Initially, we describe the tool's architecture and the core classes that should be implemented for the development of projects using the *Unreal*, both theoretically and through practical examples. Additionally, we detail the *Blueprint Visual Scripting System*, a system of visual script, which works from a mechanics based on control nodes for the development of interactive elements (game rules, players, cameras, data entry and assets). First, we show all the steps to control the behavior of a character in third person (TPS), as well as for the construction of the scene elements and materials. We then describe the procedure for creating mechanical interactions between the user and the different objects and scene elements, including the use of the particle editor, called *Cascade Particle Editor*. Finally, from the *Unreal Motion Graphics (UMG)*, a tool available in the engine, we generate a graphical interface for displaying data and basic game information (life, ammo, etc.).

[1]Programa de Pós-Graduação em Informática Aplicada (PPGIA)
Universidade de Fortaleza (UNIFOR)
{danielvalentemacedo,yvensre,andreia.formico}@gmail.com

1 Introdução

Game Engines ou motores de jogos são *middlewares* responsáveis por simplificar e abstrair elementos do desenvolvimento de aplicações gráficas ao prover um conjunto de ferramentas integradas, entre as quais: motores gráficos, de física e de conexão e rede (*network*, linguagens de *script*, *softwares* para o controle de som, gerenciadores de arquivos, etc.) [9]. A abrangência de ferramentas e funcionalidades existentes não as limita somente ao desenvolvimento de jogos, sendo amplamente usadas para a síntese de uma vasta gama de aplicações gráficas. Basicamente, a *Game Engine* funciona como uma camada entre a aplicação final e os diferentes motores e *softwares* agregados, gerando transparência ao desenvolvedor e fornecendo mecanismos eficientes de integração entre os seus componentes. A *Game Engine* é também responsável pela abstração do *hardware* final da aplicação, realizando o *deploy* específico para diferentes plataformas, tais como, computadores, *smartphones* e consoles.

Internamente, a *Game Engine* trata todos os elementos da aplicação como sendo unidades mínimas, chamadas de *game objects*, ou seja, um objeto genérico responsável por algum funcionamento dentro da aplicação (por exemplo, uma malha 3D, um controlador de efeito de som, etc.). Algumas *Game Engines* permitem a composição de *game objects* para serem reutilizados no desenvolvimento da aplicação. O funcionamento baseia-se em um *loop* nomeado *Game Loop*, o qual trata de três ações: *Entrada de Dados*; *Atualização dos Game Objects*; e *Visualização dos Game Objects*.

O *Game Loop* é executado continuamente enquanto a aplicação estiver em execução. Cada iteração no *loop* é considerado um quadro (*frame*). A *Entrada de Dados* gerencia os dispositivos de entrada (*mouse*, teclado, *joystick*, etc.) e atualiza o estado de suas variáveis, por exemplo, marcando os botões que estão pressionados ou livres durante aquele quadro. A *Atualização dos Game Objects* é responsável por iterar todos os *game objects* e chamar seus métodos de atualização, realizando todo o processamento para aquele quadro. Essa etapa costuma indicar um valor numérico para o tempo de processamento gasto entre o quadro anterior e o atual (*deltaTime*), de modo a prover um mecanismo de normalização do processamento dos elementos em diferentes *hardwares*, tendo em vista tempos de processamento diferentes. Finalmente, a *Visualização dos Game Objects* se encarrega de chamar o método de desenho dos *game objects*, quando existirem, acionando o motor gráfico para desenhar os modelos 3D na janela de visualização. Maiores detalhes sobre o algoritmo base para o *Game Loop* estão disponíveis em [9]. A *Atualização dos Game Objects* deve atualizar a lógica interna de cada *game object* e invocar métodos e funcionalidades das ferramentas da *Game Engine*. Por exemplo, *game objects* responsáveis pela emissão de efeitos de partículas e sonoros devem chamar, respectivamente, as ferramentas de partículas e de som, dentro do método de atualização. A invocação de bibliotecas como as de som é intermediada de forma a abstrair o *hardware* final do dispositivo alvo. No caso de composições de *game objects*, o método de atualização será chamado de acordo com a hierarquia da composição.

1.1 Comparação entre *Game Engines*

Basicamente, as *Game Engines* atuais variam entre si de acordo com os seguintes critérios: escalabilidade, portabilidade, ferramentas disponíveis, qualidade gráfica, custo e processamento. Mais especificamente, para o estudo comparativo apresentado neste trabalho, escolhemos três das *engines* mais populares: *Unity 3D* [5], *Unreal Engine 4* [6] e *Cryengine* [1]. A tabela 1 apresenta uma síntese desta comparação, segundo os seguintes critérios: escalabilidade, linguagem de programação principal, portabilidade (plataformas destino), curva de aprendizado e custo/mês & licença.

A *Unity 3D* apresenta a melhor curva de aprendizado e é mais leve para computadores menos robustos, embora com escalabilidade menor (não suporta simulações físicas mais realistas, a taxas interativas) e as mensalidades desestimulam desenvolvedores com fins não lucrativos. Através de *scripts*, seu método de desenvolvimento é ágil e prático, porém, voltado para programadores (e não para artistas). Atualmente, a *Unreal Engine 4 (UE 4)* é uma alternativa bastante robusta e de aprendizado mais acessível. Seu uso é gratuito, embora seja cobrado um valor de 5% sobre o lucro das aplicações que ultrapassam 3 mil dólares trimestrais, com exceção de aplicações para filmes, simulações, arquitetura e visualização [7]. Como opção, oferece acesso à programação em mais baixo nível via C++, bem como uma linguagem de programação visual robusta e de fácil aprendizado, a *Blueprint Scripting Language*, voltada tanto para programadores, quanto artistas. Já a *Cryengine* é bastante robusta e completa. Tem sido usada por vários tipos de empresas de jogos de grande porte, com alto poder de escalabilidade e alta qualidade gráfica em tempo real, suportando soluções recentes na área de iluminação e uma vasta funcionalidade para a modelagem de personagens. Contudo, sua curva de aprendizado é alta, sendo recomendada para profissionais com certa experiência. Além disso, não oferece uma documentação completa. Por outro lado, a mensalidade é mais acessível e seu contrato não envolve o pagamento de *royalties* para jogos lançados.

1.2 Detalhamento da *Unreal Engine 4*

A *UE 4* é uma suíte completa que contempla desde ferramentas para o desenvolvimento de jogos 2D para dispositivos móveis até simuladores realistas e aplicações de realidade virtual. A *UE4* também é multiplataforma, possibilitando a implementação de soluções para diferentes plataformas: Windows, MacOS, Linux, Xbox, Playstation, Steam OS, Android, iOS, etc. Oferece a alternativa de uso da *Blueprint Scripting Language*, porém, por ter um nível de abstração mais alto, produz uma sobrecarga computacional significativa em comparação ao uso do C++, sendo aproximadamente 10 vezes mais lenta [9]. Dentre suas funcionalidades, podemos destacar: *Cascade Visual Effects*, para efeitos de partícula, incluindo volume e sombra; *Material Pipeline*, para a programação de materiais; *Persona Animation*, para o gerenciamento das animações, além de gerar um mecanismo eficiente para

a programação de grafos da animação; *Matinee Cinematics*, para a produção de *cutscenes* ou filmes; *Terrain & Foliage*, para a modelagem de terrenos e flora; *Realtime Debugging*, para a depuração visual, em tempo de execução, das ferramentas integradas à suíte; e suporte ao *PhysX* em seu sistema de física, incluindo objetos articulados, simulação de tecidos e corpos rígidos, etc.

Em termos da criação de aplicações 3D, a *UE 4* apresenta uma funcionalidade diferenciada chamada *Gameplay Framework*, a qual gerencia unidades ou *Game Modes*. Um *Game Mode* corresponde à representação de um conjunto de regras para uma aplicação (número de usuários simultâneos, especificação das transições entre as *Cenas*, momento do encerramento aplicação, etc.). Essa arquitetura permite o reuso de componentes e mecânicas escolhendo-se o *Game Mode*, para alterar as regras de funcionamento da aplicação e reutilização destes elementos. A interação com os usuários é feita a partir do *Player Controller*, o qual é específico de cada *Game Mode* e obrigatório, mesmo que a aplicação não apresente nenhuma interação específica.

1.3 Apresentação da Interface da Unreal

Um projeto é criado para cada aplicação gráfica produzida na *UE 4*, sendo uma unidade auto-contida que guarda todo o conteúdo, pastas, *assets* e o código responsável pela aplicação. Quando criado ou carregado, todas as informações são exibidas na interface, como mostrado em (a) da Figura 1).

Em um projeto há, pelo menos, uma *Cena* ou *Level*, ou seja, um ambiente 3D no qual serão adicionados objetos (visuais ou não) e atores. Desta forma, podemos ter uma cena para a interface gráfica de menus da aplicação (escolher qual simulação e configurações), uma cena para cada tipo de simulação (com seus respectivos objetos e regras) e uma cena para a análise dos dados gerados.

1.3.1 Modos de Edição: Para editar e criar elementos na *Cena*, há cinco *Modos de Edição*, úteis para mudar o comportamento do editor em tarefas específicas, a saber: *Place Mode*, para adição de atores e objetos na cena; *Paint Mode*, para modificar a cor e textura dos vértices da malha estática de atores, diretamente no *ViewPort*; *Landscape Mode* para a edição de terrenos; *Foliage Mode* para a criação de instâncias de folhagem; e *Geometry Editing Mode*, para a modificação de *brushes* de geometria ((b) da Figura 1).

1.3.2 Modos de Visualização: Há diferentes modos de visualização, desde os voltados para a visualização e navegação na *Viewport*, até os de visualização dos *assets* e demais arquivos usados pela *UE 4* [10].

A *Cena* corrente é visualizada através da *ViewPort*, responsável por renderizá-la de

acordo com a configuração de gráficos escolhida pelo usuário, permitindo também a navegação no ambiente 3D com uma câmera. A *Viewport* oferece dois tipos principais de visualização: *World Outliner* e *Viewport*. Na visualização em *perspectiva*, a cena em 3D é exibida no mesmo formato da aplicação em tempo de execução; e na *ortográfica*, em 2D, é exibida nos modos de visão *Frontal, Lateral* e *Superior*. Além desses, existem modos relacionados à iluminação e renderização dos objetos: *Lit*, iluminação e tonalização em tempo real ativados; *Unlit*, iluminação e tonalização desativados; e *Wireframe* ((c) da Figura 1).

1.3.3 Navegador de Conteúdo: Todos os *assets*, códigos e demais arquivos do projeto ficam disponíveis no *Navegador de Conteúdo* ou *Content Browser*, sendo este o local primário para a criação, importação, organização, visualização e modificação destes elementos. O *Content Browser* é responsável por fornecer o gerenciamento destes arquivos com o uso de pastas e realizar operações básicas sobre estes, como renomear, realocar e criar cópias e referências às demais janelas de edição. Além disso, pode identificar arquivos que estão com problemas ou faltando, facilitando a tarefa do desenvolvedor.

1.4 Controles Básicos e Movimentação na *ViewPort*

A navegação na *ViewPort* tradicional pode ser realizada com o *mouse*, via controles de comandos de "*click and dragging*", ou através do *mouse* + teclado, via controles tradicionais de aplicações interativas em primeira pessoa. A navegação padrão com o *mouse* funciona com os comandos: `Botão Esquerdo + Drag` para mover a câmera para frente/trás e rotacionar para a direita/esquerda; `Botão Direito + Drag` para rotacionar a câmera; `Botões Esquerdo + Direito + Drag` para mover para cima/baixo. A navegação via teclado e *mouse*, também conhecida como *Game-Style Navigation*, funciona pressionando-se o `Botão Direito` do *mouse* e usando as teclas `WSAD` nas direções frente, trás, esquerda e direita, respectivamente, da mesma forma como em um aplicativo interativo em primeira pessoa. Além dos controles para navegação na *ViewPort*, é possível realizar transformações geométricas diretamente na janela, ativando-se os controles na tela, ou pelos atalhos `W` (translação), `E` (rotação) e `R` (escala). Existem ainda os seguintes controles que interferem em como a *Cena* é visualizada na *Viewport*: o *Game Mode*, tornando a renderização ativa apenas para os objetos visíveis em tempo de execução (atalho `G`); *Real Time Playback*, que ativa o *playback* em tempo real na *ViewPort* (atalho `Ctrl + G`); e *Immersive Mode*, mantém a *ViewPort* em tela cheia (atalho `F11`).

2 Aplicação 3D Interativa Desenvolvida

Nesta seção, detalharemos a aplicação 3D interativa que desenvolvemos, tanto sob o aspecto visual, quanto no aspecto técnico.

2.1 Concept e Quadros-Chaves

Desenvolvemos um jogo em terceira pessoa, no qual o usuário da aplicação controla um personagem humanóide, via teclado ou *joystick*. O objetivo é completar a travessia no cenário com o personagem, o qual pode mover-se livremente nos eixos x e z, bem como pular, interagindo com objetos e/ou escapando de ataques inimigos, ao mesmo tempo que resolve pequenos desafios. O humanóide é munido de uma arma de longa distância, capaz de atingir e destruir os seus inimigos. Estes últimos são classificados em dois tipos: destrutíveis e indestrutíveis. Ambos são fixos no chão com uma arma de *laser* apontada para uma direção fixa. Os destrutíveis podem ser destruídos por ataques do jogador, enquanto os indestrutíveis só podem ser derrotados ao serem atingidos pelo *laser* de outro inimigo.

O *Concept* é conduzir o jogador pelo cenário 3D da aplicação, exibido no mapa da Figura 2, enfrentando diversos desafios. De acordo com a Figura 2, o jogador parte do ponto "Início", em uma plataforma estática, seguindo um caminho que é interrompido nos pontos 1 e 3, preenchidos com ferramentas da *Engine*. No ponto 2, o jogador encontrará um inimigo estático. Este deve ser atacado pelo jogador e destruído, para liberar o caminho. No ponto 4 haverá mais um inimigo, desta vez, posicionado em uma plataforma móvel, dificultando o avanço do jogador. No ponto 5, apresentamos uma *cutscene* (uma área com dois inimigos indestrutíveis, estando um deles, sobre uma plataforma destrutível). No ponto 6, faremos uma apresentação das ferramentas de modelagem de terrenos e flora da *UE 4*. No 7, o jogador poderá interagir com um objeto do cenário, ativando uma plataforma elevada que o conduzirá para o "Fim" do jogo.

2.2 Programação: Objetivos e Metas

O objetivo é apresentar, detalhar e conectar, de forma prática, as diferentes funcionalidades presentes na *UE 4*. Para tal, estabelecemos as seguintes metas macros de desenvolvimento: modelagem do cenário; definição e geração da mecânica do personagem e de seus inimigos; animação do personagem e dos inimigos; síntese dos efeitos especiais de partículas; definição e uso dos materiais; e criação da *cutscene*.

3 Composição do Cenário

Os componentes básicos para a construção de um cenário na *UE 4* são: *Edição de Terrenos*, contendo diferentes métodos de criação e edição de cenários 3D; *Edição de Materiais*, apresentando a definição do *Material* e o seu funcionamento; e *Iluminação*, descrevendo os diferentes sistemas de luz e sombra usados na ferramenta.

3.1 Ambiente 3D com Editor de Terrenos e BSPs

Cada *Cena* possui um ambiente 3D, composto por primitivas geométricas já existentes na *UE 4*, luzes, efeitos visuais e modelos geométricos importados. Para a construção do ambiente usamos dois dos modos de edição: o *Place Mode* e o *Landscape Mode*.

O *Place Mode* é uma ferramenta criada para facilitar a acelerar o processo de geração de ambientes na *UE 4*, sendo bastante similar ao *Navegador de Conteúdo*, porém, focado somente nos objetos que podem ser posicionados diretamente na cena (os *Atores*). Quando ativo, o navegador do *Place Mode* mostra todos os objetos passíveis de serem adicionados no ambiente. Dentro do *Place Mode* existem os *Geometry Brushes*, úteis para construir o cenário (ou porções deste), a partir de primitivas geométricas e operações de BSP (*Binary Space Partitioning*), realizando as operações de intersecção e união entre estas primitivas.

O *Landscape Mode* é uma ferramenta usada na modelagem de ambientes realistas, tais como, cavernas, vales e montanhas. Já o seu funcionamento é parecido com os *softwares* de escultura 3D, como o *ZBrush*. O modo geral de escultura no *Landscape Mode* é através do *Sculpt Mode*, usado para trabalhar sobre o *heightmap* do cenário, moldando a sua superfície. Este último modo contém ferramentas como *Smooth* para a suavização de superfícies; *Ramp*, para seleção de duas regiões e criação de uma rampa entre as mesmas; *Erosion*, para simulação de erosão termal (por exemplo, movimentação do solo de porções mais elevadas, para porções menos elevadas); e *Hydro Erosion*, para simulação de erosão hidráulica no terreno (por exemplo, causada pela água).

3.2 Explorando o Editor de Materiais

Um *Material* é um *asset* aplicável em malhas 3D que controla os atributos de visualização dos elementos da cena, armazenando informações relativas às constantes físicas, etc ((a) da Figura 3). O uso de materiais é um detalhe importante no acabamento da cena, pois impactará no seu realismo visual. As informações armazenadas no material são usadas durante a renderização em cálculos matemáticos, a partir dos raios de luz da cena, com as características físicas do material, calculando a interação da luz com superfícies e determinando a cor de cada ponto da malha 3D que contém aquele material. A *UE 4* usa um sistema de tonalização baseado em física (*physically-based shading* ou *PBS*), para determinar as propriedades do material (difusa e especular) e características do mundo real (aspecto metálico e rugoso). Materiais também armazenam texturas, isto é, imagens 2D que variam desde uma imagem específica aplicada à superfície do material ou uma informação das normais referentes àquela superfície. Um único material pode apresentar diversas texturas para diferentes características, no entanto, as texturas são desacopladas dos materiais, podendo ser usadas por diferentes materiais e existirem, mesmo na ausência destes.

Diferente de outros elementos, os materiais não são construídos através de códigos,

mas por um mecanismo de nós (*Nodes*) visuais. Cada nó representa um trecho de código `HLSL` (linguagem de shader [10]) para uma certa tarefa. A rede de nós dos materiais funciona de modo análogo aos *scripts* em *Blueprints*. A edição da rede de nós tem como finalidade determinar os diferentes parâmetros finais de um material: *Cor Base, Cor Metálica, Cor Especular, Rugosidade* e *Cor Emissiva*. Além disso, podemos configurar características do material diretamente relacionadas às suas propriedades físicas, translucência e efeitos de pós-processamento, por exemplo.

Durante a edição da rede de nós de um material, a *UE4* permite a visualização em tempo real das alterações feitas no material de duas formas: *Live Nodes*, na qual informações constantemente modificadas (como variações nos eixos de coordenadas) são atualizadas em tempo real; e *Live Update*, na qual há a recompilação de cada trecho do *shader*, sempre que uma alteração é feita, atualizando-se a visualização.

3.3 Iluminação do Ambiente

3.3.1 Tipos de Luzes Existem diferentes formas de adicionar luz em uma *Cena* e diferentes tipos de luzes ((b) da Figura 3). A adição de um ponto de luz é feita no editor de *level*, inserindo-se: *Point Light*, um ponto de luz móvel que emite luz em um determinado volume, em todas as direções; *Spot Light*, um foco de luz direcionável e móvel que emite luz em um volume em uma direção específica; e *Directional Light*, um ponto de luz fixo, localizado no infinito que emite luz em toda a *Cena*, a partir de uma direção específica. Cada ponto de luz possui diferentes características: (1) *Intensidade*, ou quantidade de energia que o foco de luz emite na cena (*Point Lights* e *Spot Lights*, com intensidade de 1700 lumens, os quais correspondem a lâmpadas reais de aproximadamente 100W); (2) *Cor*, escolhida no editor; e (3) *Raio de Atenuação*, para representar o alcance da luz e quais objetos são atingidos por ela. Vale ressaltar que a informação do alcance possui um alto impacto no desempenho da aplicação porque aumenta exponencialmente a quantidade de cálculos realizados durante a renderização.

3.3.2 Iluminação Pré-Processada ou Light Mass Baking Cada foco de luz pode ser classificado como: *Static*, completamente estática; *Stationary*, que pode mudar de cor e intensidade durante a execução, mas incapaz de realizar transformações geométricas; e *Movable*, completamente dinâmica, capaz de mudar todas as suas propriedades em tempo real.

Tão importante quanto a iluminação de uma cena é a geração de sombras. Na *UE4*, sombras são calculadas de uma forma diferente, dependendo do tipo de luz que interage com os componentes do cenário. Sombras formadas por luzes estáticas têm custo computacional mais baixo, ao contrário das luzes dinâmicas, que precisam ser recalculadas a cada quadro. Existem quatro tipos de sombras na *UE4*: *Directional Light Cascading Shadow Maps*; *Stationary Light Shadows*; *Dynamic Shadows*; e *Preview Shadows*.

Directional Light Cascading Shadow Maps ou *Whole Scene Shadows* são sombras estáticas calculadas para toda a cena através de *Cascades Shadow Maps* [8], junto às sombras criadas a partir de iluminações estáticas. Esse tipo de sombra é muito útil em cenários repletos de objetos estáticos, como árvores e folhagens, nos quais as sombras são praticamente estáveis. *Stationary Light Shadows* são sombras calculadas individualmente para cada objeto dinâmico. As sombras são calculadas e integradas no sistema global de sombras através de testes de distância da câmera e seus respectivos focos de luz. O custo dessas sombras depende da quantidade de objetos dinâmicos existentes na cena. *Dynamic Shadows* são sombras sintetizadas a partir de focos de luz dinâmicos (calculados para todos os objetos da cena), sendo, dentre os tipos de luzes citados, o mais custoso. *Preview Shadows* são sombras geradas no período de produção de uma *Cena* para evitar o cálculo das sombras, repetidas vezes. Em síntese, são armazenadas pelo editor e usadas durante a edição da *Cena* enquanto a aplicação não está em execução. Sua principal funcionalidade é oferecer ao desenvolvedor uma ideia de como as sombras reais aparecerão no quadro gerado em tempo de execução, de forma ágil, embora menos fiel e realista.

4 Interatividade e Visualização

A modelagem da interatividade e da mecânica da aplicação apresentada é feita via *Blueprint Script*, usando três tipos de dispositivos de entrada: teclado, *mouse* e *joystick*.

4.1 Configuração dos Controles de *Input*

Para o tratamento de *Input*, há a classe *PlayerInput* (*UObject*), disponível no *PlayerController*, responsável pelo tratamento dos *inputs* do usuário. Dois tipos de mapeamentos podem ser criados nesta classe: (1) *ActionMappings* e (2) *AxisMappings*. O primeiro faz o mapeamento das ações de "pressionar" e "soltar" as teclas do teclado e os botões do *mouse* ou do *joystick*, mapeando uma tecla do teclado e um botão do *joystick* para executar um mesmo evento, facilitando, por exemplo, o desenvolvimento de uma aplicação multiplataforma. O segundo mapeamento gera eventos a todo *frame* e é usado para tipos de entrada nos quais é preciso verificar uma variação de valores a todo momento (e não somente se um botão foi pressionado ou não). A alavanca do *joystick* é um típico exemplo, já que dependendo da intensidade do movimento, pode-se incrementar a velocidade de um elemento da aplicação, usando uma mesma proporção. A configuração dos *Inputs* do projeto é feita na *Unreal* em Edit → Project Settings → Engine → Input [7].

A *Unreal Engine* agregou um benefício diferenciado para um grupo mais amplo de desenvolvedores de aplicações gráficas interativas, substituindo o *UnrealScript* (UDK) pelo *Blueprint Visual Script*, voltado para usuários não conhecedores de código e linguagens de

programação. Essa linguagem possibilita o desenvolvimento de diversas mecânicas e funcionalidades, em um nível de abstração alto, baseado em nós de controle. Em contrapartida, produz uma sobrecarga computacional significativa em comparação ao uso da linguagem C++, já que funciona como uma camada de mais alto nível, localizada sobre as camadas de código da *Unreal*. Ao criar um *Blueprint*, é preciso definir qual classe deverá ser herdada. Por padrão, a *Unreal* apresenta as seguintes opções: (1) *Actor*: qualquer objeto adicionado ao mapa (cenário), possuindo suporte para transformações geométricas 3D (translação, rotação e escala), sendo que *Actors* podem ser criados e destruídos durante o *gameplay*; (2) *Pawn*: classe base de todos os *Actors*, controlados por usuários ou IA, sendo que um *Pawn* é a representação de um usuário ou entidade de IA dentro do mapa; (3) *Character*: tipo de *Pawn* que já inclui a habilidade de movimento no mapa; (4) *Player Controller*: um *Actor* responsável por controlar um *Pawn* que pertence a um usuário; (5) *Game Mode*: regras que definem a lógica da aplicação gráfica sendo executada; (6) *Actor Component*: elemento que pode ser adicionado a qualquer *Actor*; (7) *Scene Component*: equivalente a um *ActorComponent*, porém, possui propriedades de transformação geométrica e pode ser acoplado a outros componentes.

O *Blueprint Editor* possui três janelas principais: *Viewport*, *Construction Script* e *Event Graph*. Na *Viewport* é feita a construção do *Blueprint* com vários componentes diferentes. O *Construction Script* é um evento especial, executado toda vez que o *Blueprint* sofre alguma alteração, muito útil para criar *Blueprints* de objetos que compõem um mapa (por exemplo, gerar uma cerca com tamanho variável). No *EventGraph* é realizada a adição da lógica do *Blueprint* via *Blueprint Visual Script*.

A programação no *Event Graph* funciona basicamente a partir de eventos e chamadas de nós de controle que representam, normalmente, funções em C++ que são expostas aos *Blueprints*. Um evento (representado na interface como um nó vermelho), é o ponto de partida de cada trecho de código. A partir dele, a *Unreal* irá executar sequencialmente a cadeia de nós que serão executados e estarão conectados pelas ligações que saem das setas (representadas na interface na cor branca). Para os parâmetros de entrada e saída dos nós, a *Unreal* define um padrão de cores, de tal forma a facilitar a leitura e identificar os tipos de dados em uso. Inicialmente, a *engine* já adiciona no corpo do *Event Graph*, três eventos básicos: *Begin Play* (executado quando a aplicação é iniciada), *Actor Begin Overlap* (ao colidir com outro actor) e *Tick* (executado a cada frame).

O *Blueprint* oferece praticamente todos os recursos que uma linguagem de programação tradicional possui: operações aritméticas, comparações, *loops*, declaração de váriáveis e funções, etc. Além disso, é possível implementar uma classe puramente C++ e estendê-la usando *Blueprints*, *pipeline* bastante recomendado pela *Epic Games*.

4.2 Programação das Mecânicas

Nesta seção, detalharemos como foram implementados os componentes que compõem o projeto desenvolvido.

4.2.1 *Blueprint* do Personagem Principal Para o personagem principal, criamos um *Blueprint* a partir da classe *Character*, o qual já possui vários comportamentos de movimento e física implementados. Na lista dos componentes, adicionamos um *Spring Arm Component*, responsável pelo movimento da câmera e um *Camera Component*, que corresponde à câmera principal e filha do *Spring Arm* (que já possui implementado alguns comportamentos que evitam que a câmera penetre em objetos da cena). Para compor a malha do personagem, utilizamos o *Skeletal Mesh* de terceira pessoa do herói que faz parte do exemplo *ShooterGame* da própria *Epic Games*, disponível gratuitamente para *download* através do *launcher* ((a) e (b) da Figura 4).

4.2.2 Movimentação Para o movimento de rotação do personagem foram usados os *inputs LookUp* e *LookRight* que rotacionam, respectivamente, o *pitch* e o *yaw* da câmera, de forma similar ao jogo *Dead Space* [2]. Para o movimento de translação, foram mapeados os *inputs MoveForward* e *MoveRight*, que movem o personagem, respectivamente, em direção ao seu *forward vector* e ao seu *right vector*.

4.2.3 Gerenciando as Animações Para controlar as animações do personagem, foi criado um *Animation Blueprint*, que é um tipo de *Blueprint* especial, responsável por comandar as animações de um *Skeletal Mesh*. No *Animation Graph*, criamos uma máquina de estados para o personagem: *Run, JumpStart, JumpLoop* e *JumpEnd* (Figura 5). Inicialmente, o personagem está no estado *Run* que possui uma animação do tipo *BlendSpace*, na qual as animações de correr para frente, trás, esquerda, direita e o *idle* são interpoladas, gerando transições, de acordo com os parâmetros *Direction* e *Speed*, obtidos a partir do *Event Graph*. Mais especificamente, se o personagem está só se movimentando, se mantém no estado *Run*. Ao pressionar a ação de *input Jump*, o personagem deixa de encostar no chão, é feita então uma transição para o estado *JumpStart*, que ativa a animação de pulo e, logo em seguida, quando a animação está em 30% para finalizar, uma nova transição é feita para o estado *JumpLoop*, até que volte a encostar no chão, fazendo uma transição para o *JumpEnd* e, em seguida, para o estado *Run*. Para complementar o movimento, uma outra animação com um *BlendSpace*, representando as posições extremas da mira é combinada de forma aditiva e enviada para o resultado final da animação. Dessa forma, o personagem consegue correr e pular, ao mesmo tempo que mira em uma certa direção.

4.2.4 Armas e Tiros Para a arma, criamos um *Actor Blueprint* com o lançador de granada, também importado do exemplo *ShooterGame* da *Epic Games*. Um *Scene Component*

foi adicionado para representar uma posição no espaço referente à ponta do cano da arma, indicando onde a partícula que compõe o efeito do tiro deve ser instanciada, chamada de *WeaponFirePoint*. Definimos uma variável do tipo inteira para armazenar o total de balas da arma e uma função *Fire*, chamada pelo personagem ao ativar o evento de *input* de *Fire*. Na chamada da função do tiro, a partícula que representa o *nozzle* da arma é instanciada sobre o *Scene Component* que está posicionado na ponta da arma. Ao reduzir o número de balas, ativamos o som do tiro, e um *LineTrace* é disparado para verificar se houve alguma colisão na direção de tiro (centro da tela). Em caso afirmativo, uma outra partícula de explosão é instanciada exatamente no ponto de colisão identificado pelo *LineTrace*. Para posicionar a arma na mão do personagem, criamos um *socket* na *Skeletal Mesh* da mão do personagem, conhecido como *WeaponPoint*. No evento *Begin Play*, o *Blueprint* da arma é instanciado e acoplado ao *socket,* fazendo com que o personagem segure a arma.

4.2.5 Munição Para a munição, geramos um *Actor Blueprint* com um *Static Mesh* de uma caixa. Adicionalmente, criamos uma variável do tipo inteira para indicar quantas balas existem no item de munição e configuramos o *Collision Presets* para *OverlapAll*. Assim, o movimento do personagem não é interrompido ao passar pela munição. Já para a coleta da munição, adicionamos ao evento *Actor Begin Overlap* do personagem uma verificação para testar se o objeto é um *Blueprint* de munição, ou não. Em caso afirmativo, somamos o total de balas da caixa de munição ao total atual da arma que o personagem está segurando. Finalmente, removemos o *Blueprint* da munição do mapa.

4.2.6 Radar de Objetivos Para criar o radar de objetivos, adicionamos ao cenário um *NavMeshBoundsVolume*, responsável por calcular caminhos entre dois pontos no mapa. Após posicionar o volume corretamente, geramos um *Actor Blueprint* que consulta o *NavMesh-BoundsVolume* para obter uma lista de pontos que indica o menor caminho entre a posição atual do personagem e o seu próximo objetivo. Com essa lista, criamos uma *spline* usando um *Spline Component*, para criar um caminho que passe por esses pontos. Em seguida, instanciamos *Spline Mesh Components*, que corresponde a um tipo de *mesh* deformável.

4.3 Elementos do Cenário e Controle de Eventos

4.3.1 Plataformas Móveis Para o movimento das plataformas, geramos uma *Timeline* com uma curva do tipo *float* que varia de 0.0 até 1.0 e de 1.0 até 0.0 em *loop*, ativado no evento *Begin Play* do *Actor Blueprint* da plataforma. Para fazer o movimento, multiplicamos o valor do ponto atual da curva por um *vector offset* (que representa a posição final do deslocamento da plataforma) e somamos a posição inicial da mesma, a cada atualização da *Timeline*.

4.3.2 Inimigos Para os inimigos, criamos variações de um *Actor* com *MatPreviewMesh* do *ContentPack* que já vem com a *UE 4*. Os inimigos disparam *lasers* gerados por efeitos de partículas pelo logo da *UE 4*, em tempos variados ou constante. Alguns são impossíveis de serem destruídos e outros, ao serem destruídos por tiros do lançador de granada, sofrem um explosão e são destroçados. Para quebrar a malha em pedaços, usamos o recurso de *Destructible Mesh* da *UE 4*, que é capaz de fragmentar a malha em vários segmentos, como mostrado na Figura 6.

4.3.3 Eventos com *Waypoints* Para disparar eventos e detecção de *waypoints*, distribuímos *Actor Blueprints* com *Box Collision*, ou seja, com somente um volume invisível que dispara eventos de *Actor Begin Overlap* (disparado quando algum objeto passa por dentro dele). Com isso, quando o personagem atravessa essas caixas invisíveis, o *GameMode* é acionado para informar que este *waypoint* foi atingido.

4.4 Criação de Efeitos com o Editor de Partículas

Toda a criação de efeitos de partículas é feita a partir de um editor chamado de *Cascade*, com o qual é possível adicionar um ou mais *Emitters*, componentes responsáveis por emitir partículas. Cada *Emitter* é composto por *Modules*, que determinam vários comportamentos da partícula, tais como: cor, tamanho, tempo de vida, velocidade, entre outros. Para compor o efeito de explosão que modelamos (quando o tiro do lançador de granada atinge um alvo), cinco *Emitters* distintos foram usados para simular os efeitos de: (1) onda de choque; (2) bola de fogo; (3) faísca; (4) iluminação e (5) fumaça, como mostrado na Figura 7.

4.5 Adição de *Cutscene* com a Ferramenta de Animação (*Matinee*)

A *UE 4* possui um editor de *cutscenes* e animações, o *Matinee*, voltado para animar as propriedades de *Actors* ao longo do tempo, criar dinâmicas de *gameplay* e *cutscenes*. O sistema baseia-se no uso de *tracks* de animação nas quais é possível incluir quadros chaves para definir os valores de determinadas propriedades dos *Actors* no mapa. O *Editor* é semelhante aos editores de vídeo populares, tornando-se bastante familiar para os profissionais já acostumados com outros editores. Para cada *cutscene* do nosso exemplo, existe um *Matinee Actor*, disparado por eventos a partir dos *waypoints*, contendo uma *track* com os quadros chaves das posições e enquadramento que a camera seguirá durante toda a animação.

4.6 Adição de Interface Gráfica Básica (*Unreal Motion Graphics*)

Para criar o HUD usamos o UMG (*Unreal Motion Graphics*). No UMG adicionamos uma imagem do lançador de granada no canto superior direito e uma *progressbar* com o

modo de desenho de imagem para apresentar uma imagem de até 10 balas, parcialmente, de acordo com a munição atual da arma. Temos também no centro superior da tela um contador que indica o tempo restante para finalizar todos os objetivos do exemplo desenvolvido e, no canto esquerdo, uma *progressbar* simples indicando a vida. O *widget* do HUD é adicionado na *viewport* pelo *GameMode* no evento *Begin Play*.

4.7 Finalização e Deploy da Aplicação

Para fazer o *deploy* da aplicação, devemos passar pelo processo de *Packaging*. O processo de *packaging* garante que todo o código e conteúdo esteja preparado e no formato adequado para ser executado na plataforma desejada. O processo de *packaging* consiste em um conjunto de passos. Por exemplo, quando um projeto é feito em C++, este código primeiramente é compilado. Em seguida, todo o conteúdo (ou *Content Cooking*) tem que ser convertido para um formato utilizável na plataforma alvo. Finalmente, o código compilado e o conteúdo convertido são empacotados de acordo com a plataforma escolhida para o *deploy*. O processo de *packaging* pode ser executado em File $\rightarrow$ Package Content $\rightarrow$ TARGET (Ex: Android, iOS, HTML5, Linux, Mac e Windows). É importante ressaltar, que para o deploy para HTML5, é preciso ter instalado na máquina o *Emscripten* [3] e o *Python 2.7* [4].

5 Conclusão e Trabalhos Futuros

Apresentamos detalhadamente, em termos teóricos e práticos, os principais conceitos para o desenvolvimento de aplicações gráficas interativas em tempo real usando a *UE 4*. Mostramos os passos para o controle do comportamento de um personagem em terceira pessoa, para a construção dos elementos e materiais do cenário, para a criação de mecânicas de interação entre o usuário e os diferentes objetos e elementos e geramos uma interface gráfica para a apresentação dos dados e informações básicas do jogo. Além disso, mostramos o potencial da ferramenta como uma solução alternativa para o desenvolvimento de uma vasta gama de aplicações, ressaltando a presença de bibliotecas modernas de Física, *workflow* integrado de ferramentas e baixa curva de aprendizado, acessível não somente para profissionais da Computação, via *Blueprint Scripting Language*. Ao tornar o seu código público, a *UE 4* permitiu que grandes estúdios de produção de jogos criassem extensões próprias da ferramenta, tornando-a mais competitiva frente a outras *game engines* similares existentes na atualidade. Por outro lado, há também certas funcionalidades ainda não contempladas pela *UE 4*, tais como, simulação de fluídos, bem como algumas limitações existentes importantes, por exemplo, os requisitos mínimos de sistema exigidos para o bom funcionamento da *UE 4*, que impossibilitam o uso da *game engine* em dispositivos com recursos computacionais menos robustos.

Como trabalhos futuros, a *UE 4* tem mostrado interesse no mercado cinematográfico, visando a produção de filmes gerados por computador em tempo-real. Atualmente, os filme por computador usam renderizadores *offline* e, em geral, levam várias semanas para a produção final, usando dezenas de máquinas em paralelo. Nessa linha, a *Epic Games* desenvolveu o projeto *Kite*, um curta-metragem cujo cenário é bastante realista, com duração de aproximadamente 2 minutos, executando em um único computador com uma GTX Titan X, a uma taxa de 30 quadros por segundo. Adicionalmente, a *UE 4* promete ainda incluir outras funcionalidades, por exemplo, renderização de efeitos para representação de cabelos e pêlos, suporte para a destruição dinâmica de objetos em tempo real, para gestos do tipo *touch screen* e para conexão p2p, *launcher* para Linux, além de iluminação simplificada, sombras dinâmicas e decaimento para *tablets* e *smartphones*.

Agradecimentos

Daniel Valente Macedo gostaria de agradecer à CAPES e Yvens Rebouças Serpa à FUNCAP-CE (Processo PEP-0094-00005.01.09/2014), pelo auxílio financeiro recebido.

6 *

Referências

[1] Crytek CryENGINE 3. http://www.crytek.com/cryengine. Último acesso em: 22/05/2015.

[2] Dead Space Game. http://www.ea.com/deadspace/. Último acesso em: 25/05/2015.

[3] Emscripten. https://kripken.github.io/emscripten-site/. Último acesso em: 25/05/2015.

[4] Python. https://www.python.org/download/releases/2.7/. Último acesso em: 25/05/2015.

[5] Unity 3D Engine. https://unity3d.com/pt. Último acesso em: 22/05/2015.

[6] Unreal Engine 4. https://www.unrealengine.com/. Último acesso em: 22/05/2015.

[7] Unreal Engine 4 Documentation. https://docs.unrealengine.com/latest/INT/. Último acesso em: 22/05/2015.

[8] W. Engel. *Cascaded Shadow Maps. ShaderX5, Advanced Rendering Techniques*. Charles River Media, 2006.

[9] Jason Gregory. *Game Engine Architecture*. CRC Press, 2009.

[10] Randi J Rost, Bill M Licea-Kane, Dan Ginsburg, John M Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL Shading Language*. Pearson Education, 2009.

Tabela 1. Tabela Comparativa: *Unity 3D*, *Unreal Engine 4* e *Cryengine*.

	Unity 3D	***Unreal Engine 4***	***Cryengine***
Escalabilidade	Média	Alta	Alta
Linguagem de Programação	*C#, Javascript, Boo*	*C/C++* e *Blueprint*	*Lua Scripting 4*
Portabilidade	*Mobile*, *Desktop* e *Consoles*	*Mobile, Desktop* e *Consoles*	*Mobile, Desktop* e *Consoles*
Curva de Aprendizado	Fácil	Média	Alta
Custo/mês & Licença	Gratuita e Mensalidade de US$ 75,00	Gratuita e OpenSource com 5% de royalties	Mensalidade US$ 9,90

(a) Interface principal.

(b) Modos de edição.

(c) Modos de visualização: *Lit, Unlit* e *Wireframe*.

Figura 1. Detalhes gerais da Interface da *Unreal Engine 4*.

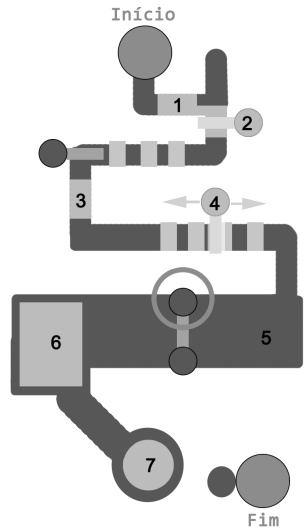
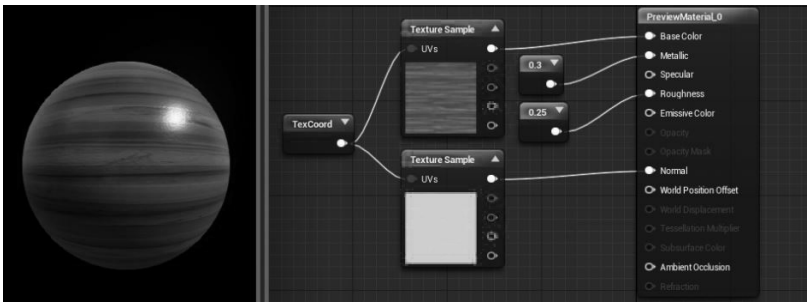


Figura 2. Mapa do cenário da aplicação exemplo.

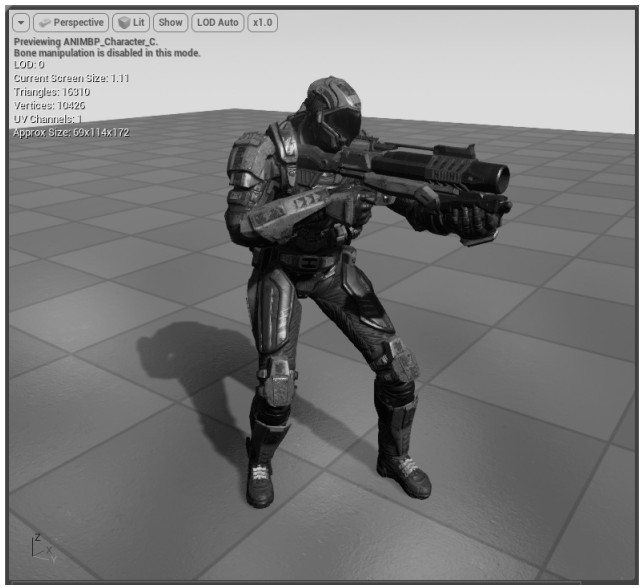


(a) Exemplo de Material e a Rede de Nós para a construção de materiais.

(b) Tipos de Luz: Point Light, Spot Light e Directional Light.

Figura 3. Detalhes relativos aos materiais e iluminação na *UE 4*.

(a)



(b)

Figura 4. *Blueprints* do personagem principal.

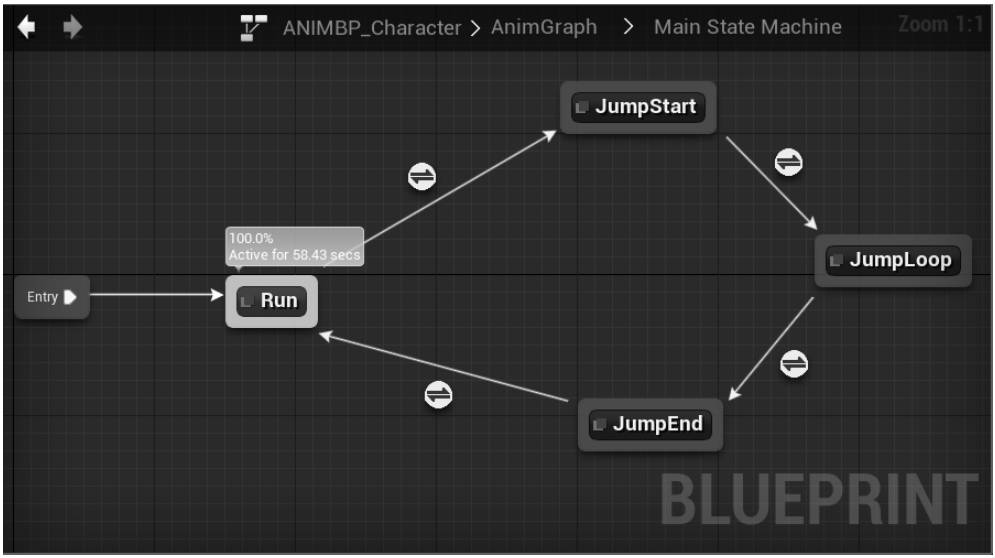


Figura 5. Máquina de estados das animações.

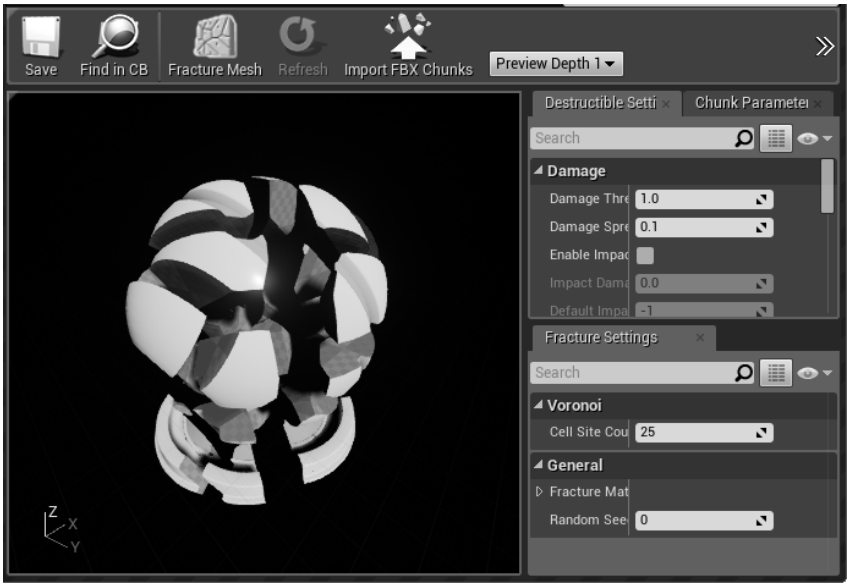


Figura 6. Recurso de *Destructible Mesh*.

Figura 7. *Cascade* e efeitos de partículas.