Desenvolvendo aplicativos e games usando o ARKit

Mark Joselli*

PUCPR, Apple Developer Academy, Brasil

RESUMO

Com o aumento do poder de processamento junto com o aumento dos sensores atuais dos dispositivos moveis e com o desenvolvimento de frameworks, torna-se possível a utilização de aplicativos de realidade aumentada nesses dispositivos. O maior desafio neste tipo de aplicativo é introduzir objetos gerados pelo computador no mundo real capturado através de um câmera, de forma que fique dentro do contexto do mundo real, isto é, alinhada e interagindo com objetos do mundo real. Em games, existe ainda mais um complicador: a mecânica. A mecânica do jogo deve ser adaptada para esse novo tipo de interação. O desenvolvimento de aplicativos de AR pode ser realizado através do uso de diversos frameworks existentes no mercado. Neste capitulo será utilizado o ARKit, um framework desenvolvido pela Apple, gratuito e simples de utilizar. Para realizar uma demostração este capitulo se utilizará do XCode, ambiente de programação para dispositivos Apple, e a Unity, motor de jogo para diversas plataformas.

Palavras-chave: iOS, Swift, SpriteKit, SceneKit, Metal, Unity e ARKit.

1 INTRODUÇÃO

Aplicações de realidade aumentada (AR (Augmented Reality)) tiveram como limitante o hardware no qual as mesmas eram executadas, já que este tipo de aplicação exige um alto poder de processamento em tempo real. Neste tipo de aplicação, existe a necessidade de renderização e rastreamento dos objetos reais para objetos virtuais, sendo esta última operação bastante onerosa do ponto de vista de processamento. Aplicações em máquinas Desktop apresentavam, na maioria das vezes, a melhor performance para execução deste tipo de aplicação. Porém, tais sistemas apresentam a ausência de um aspecto muito importante neste tipo de aplicação: mobilidade. A maioria de aplicações necessitavam da manipulação da câmera, muitas vezes sendo este um entrave para sua maior popularização. Devido à grande presença dos dispositivos móveis e estes apresentando um poder de processamento cada vez maior, além de uma significativa melhora em suas câmeras (componente essencial para a AR), sua utilização para processamento de aplicações em realidade aumentada vem crescendo cada vez mais. Atualmente é possível acessar vários aplicativos de realidade aumentada nas markets stores de cada sistema operacional, seja este iOS ou Android.

Este capitulo tem o objetivo de apresentar o ARKit framework de realidade aumentada disponível para os dispositivos iOS. No ultimo evento da Apple, o WWDC, foi apresentado os novos recursos do iOS 11 e dentre eles esse novo framework que prove uma seria de inovações de forma a criar aplicativos com realidade aumentada.

Diversos framework estão disponíveis no mercado, com o objetivo de facilitar o desenvolvimento deste tipo de aplicação, sendo alguns destes gratuitos e outros comerciais. O ARKit utilizado durante o capitulo é gratuito, apresentando uma implementação eficiente de algoritmos para rastreabilidade de objetos em tempo real, possibilitando a interação do ambiente real e o virtual sem perder a noção de interatividade. Com isso abre-se um novo campo para a aplicação de realidade aumentada em games, que diferente dos games tradicionais, existem novos desafios e possibilidades de diversão para serem desenvolvidos. O framework apresenta suporte para dispositivos baseados em iOS, que podem usar o SpriteKit, o SceneKit ou o metal para a renderização. Outra opção é o motor de jogo Unity3D e Unreal que podem ser usado com plugins que encapsula as principais funcionalidades do framework.

A realidade aumentada é uma nova experiência que prove a funcionalidade de colocar elementos 2d ou 3d dentro da câmera do dispositivo de uma forma que esses elementos aparentam fazer parte do mundo real. O ARKit combina diversas tecnologias para facilitar o processo de construir uma experiência AR, como monitoramento do movimento do dispositivo, captura da cena da câmera, processamento dessa cena e facilidades de display.

O ARKit se utiliza de Visual Inertia Odometry (VIO), ou traduzindo Odometria Inercial Visual, para realizar o monitoramento do mundo capturado pela câmera. Esse VIO combina a captura da câmera com os dados do movimento do Coremotion, com dados do acelerômetro, giroscópio, pedômetro, barômetro e magnetometro. Com isso o dispositivo pode colocar AR em um ambiente com alta fidelidade, sem calibração adicional.

Além disto, o ARKit analisa a cena apresentada pela câmera e detecta planos horizontais de forma automática na cena, como mesas ou o chão. Com isso, objetos virtuais podem ser inseridos no topo desses planos. ARKit também se utiliza do sensor de luminosidade da câmera, de forma detector o grau de luminosidade da cena real e replica na cena virtual. O ARKit é altamente otimizado para os processadores A9 e A10 da apple de forma a disponibilizar cena em AR com alta performance. Para disponibilizar as cenas, o desenvolvedor pode se utilizar de SpriteKit, Metal, SceneKit ou até mesmo Unity.

Este tutorial pretende, inicialmente, explicar o funcionamento da tecnologia utilizada por aplicações de realidade aumentada. Finalmente pretende-se mostrar a instalação do ARKit e criação de um projeto. Além disto um pequeno game será desenvolvido, juntamente com o grupo, onde se acompanhará desde o processo de design até a construção do código fonte para rodar.

Devido as diferentes características de AR, como o uso de câmera, e dos dispositivos móveis, que além da possibilidade de AR, oferecem outros recursos, como localização e diferentes tipos de input [4, 1], games que usam esta tecnologia devem ter um game design com um paradigma diferente que os jogos com o input tradicional.

Este tutorial pretende, inicialmente, explicar os conceitos básicos da tecnologia utilizada por aplicações de realidade aumentada na seção 2. Na seção 3 é abordado o framework do ARKit, mostrando seus principais conceitos e utilização, bem como as tecnolgias que auxiliares usadas para a renderização. A seção 4 descreve a parte prática utilizada neste tutorial, onde pretende-se mostrar a utilização do ARKit e criação de um projeto com AR utilizando-se do XCode, seguido da seção 5, que mostra a Unity em conjunto com o ARKit. Finalmente a seção 6 apresenta a conclusão do trabalho.

*e-mail: mark.joselli@pucpr.br

2 REALIDADE AUMENTADA

A realidade aumentada, muitas vezes também denominada de AR (Augmented Reality) consiste em incrementar imagens do mundo real com elementos virtuais. Estes elementos virtuais podem ser gerados sinteticamente ou podem ser objetos pré-processados ou pré-armazenado, tais como imagens, fotos ou até mesmo trechos de vídeos. Diferentemente da Realidade Virtual, onde se procura criar uma realidade alternativa de forma totalmente digital, a AR procura usar como base a própria realidade sendo capturada por algum dispositivo, tal como câmeras fotográficas ou filmadoras [7]. São inúmeras as aplicações possíveis para esta área, tais como arquitetura, arte interativa, comércio, construção, medicina, educação e jogos.

Podemos separar a AR em duas categorias: informativo e Imersivo. A AR informativa é em geral mais simples e basicamente consiste em acrescentar a uma imagem que está sendo capturada dados relevantes para serem mostrados. Entretanto, em muitos casos, deseja-se mostrar os dados em locais específicos da imagem ou associadas ao contexto, tais como nomes de ruas, informações de um veículo ou de um local. Para tanto, é necessário realizar tratamentos de imagens que permitam extrair informações semânticas. A AR imersiva contém elementos da AR informativa, porém além de acrescentar dados, serão acrescentados elementos virtuais tentando mesclar-se com os elementos reais.

O tratamento da AR imersiva pode ser bastante complexo, pois exige que seja feito preliminarmente um reconhecimento do contexto da imagem sendo capturada, de forma a fazer uma correta composição. Para realizar este processo, utiliza-se em geral técnicas de visão computacional, que irão separar e identificar objetos e elementos da cena [2, 3]. De forma geral, os algoritmos de visão irão realizar algum tipo de processamento da imagem em conjunto com um processo de segmentação, que basicamente consiste em detectar bordas e contornos, tentando assim separar cada elemento. Uma vez extraídas as características das imagens, realiza-se o reconhecimento de padrões, onde se tentará localizar elementos ou características pré-definidas de algum dado. Nesta etapa podem-se usar classificadores e/ou métodos de comparação com bases de dados já existentes. Outro tarefa importante a ser resolvida pela AR imersiva consiste em calcular uma estimativa da posição da câmera real. Isto deve ser feito, pois ao colocar objetos virtuais, uma câmera virtual também deve ser criada, coincidindo com as mesmas coordenadas, movimentos e condições da câmera virtual.

Embora o dispositivo básico para captura da AR seja uma câmera, outros dispositivos podem ser importantes para o mapeamento do movimento ou posicionamento da mesma. Assim, sensores de GPS, acelerômetros e giroscópios podem facilitar o trabalho de mapeamento e detecção de posicionamento de câmera. Para mostrar o resultado da AR podem haver diversas interfaces, podendo ser desde uma tela LCD até head-mounted displays (HMD), eyeglasses ou projetores.

O ponto chave para o sucesso de uma aplicação imersiva de AR é o quão convincente é o resultado de uma composição do virtual com o real. Em muitos casos não é essencial que os elementos virtuais sejam extremamente realistas, a ponto de confundir-se com as imagens reais, mas é fundamental que a composição esteja correta e os dados virtuais estejam acertadamente colocados sob os reais. Para isto, é fundamental que os métodos de visão computacional sejam capazes de acertar na detecção e separação dos objetos.

De forma geral, não é necessário detectar as coordenadas de todos os objetos da cena, mas apenas de alguns que serão importantes na composição. Em algumas aplicações onde a detecção de objetos deve ser mais preciso, é possível armazenar previamente informações referentes a cena, de forma que o processo de visão tenha mais informações de mundo. Esta técnica é particularmente conveniente em casos onde o cenário sendo capturado é bem conhecido, tal como um estádio de futebol ou uma cabine de avião.

2.1 Jogos com ARKit

Jogos podem utilizar-se de recursos de realidade aumentada na sua concepção. Embora seja uma ótima forma de aumentar a percepção de interação e imersão, é fundamental que jogos nesta categoria não priorizem a inserção de elementos de AR em vez de focar nos aspectos de jogabilidade em si. De fato, diversos jogos desenvolvidos até o momento dentro desta categoria falharam como produtos, por colocar em primeiro lugar a mecânica da AR e não a ludicidade em si.

Neste sentido, o planejamento de qualquer jogo deve ter origem na concepção de suas regras. São estas regras, que bem calibradas e medidas, garantem a diversão de um jogo. De acordo com Koster [9], a diversão provocada por um jogo consiste em padrões apresentados ao cérebro, que ao sentir-se desafiado, tenta imediatamente resolver e construir um processo mental capaz de ?dominá-lo?. Quando este padrão é entendido, ocorre um processo de recompensa, que provoca um bem estar e portanto um momento lúdico. Quando este desafio é complexo demais e o padrão não consegue ser entendido, o cérebro cria uma repulsa, tal como quando vemos uma imagem ruidosa que não somos capaz de entender o que vem a ser. Por outro lado, quando o desafio é muito simples e o cérebro não precisa construir nenhum processo mental, esta recompensa não ocorre, não havendo também o bem estar lúdico.

Este desafio ou padrão pode ser apresentado metaforicamente, através de uma estória, personagens e cenários conhecidos. Entretanto, o cérebro internamente irá criar um processo de abstração de forma a focar-se no desafio em si. Por esta razão, narrativas e interfaces imersivas podem ser interessantes, mas não fundamentais. São interessantes porque podem facilitar o processo mental para entender a mecânica que se quer desafiar ao cérebro. Assim, da mesma forma que quando no projeto de game design se prioriza uma narrativa ao invés da mecânica, o jogo tende a tornar-se menos interessante e potencialmente menos lúdico, o mesmo pode ocorrer quando se prioriza a interface. Isto é particularmente relevante quando se trata de um jogo com AR, já que normalmente todas as aplicações neste cenário requerem uma mecânica de reconhecimento de padrões, calibração de câmera e de certa forma utilizar um cenário controlado.

De fato, uma das maiores restrições tecnológicas para os jogos com AR consiste no cenário pré-definido ou pré-concebido que deve haver, em função do reconhecimento de padrões necessário na etapa de visão computacional. Desta forma, embora se sonhem com jogos com AR de FPS (First Person Shooters), onde se pode caminhar por um mundo real e inserir personagens e elementos virtuais, na prática, estes não são bons exemplos de jogos atuais que usam AR. Assim, jogos de tabuleiros ou jogos onde códigos QR não interfiram no gameplay, são os primeiros candidatos a poder explorar bem esta tecnologia. Como de forma geral é importante ter mobilidade com uma câmera, plataformas móveis se adequam melhor a jogos com AR. Um exemplo do jogo StarWars HoloChess com ARKit pode ser visto na Figura 1.

3 ARKIT

O framework do ARKit framework é uma plataforma de realidade aumentada que permite o desenvolvimento de aplicações que fazer uso de AR em dispositivos moveis. Ele é uma poderosa APT mas com uma interface simples, de forma que com um pouco de conhecimento técnico se consegue fazer apps impressionantes. Como o ARKit requer recursos especiais de hardware, ele somente esta disponível em dispositivos com o processador A9 ou superior. O ARKit tem as seguintes capacidades:

- Posicionamento: ele consegue através do odômetro inercial, combinado com os sensores de movimento e com a câmera ter o posicionamento do dispositivo no mundo real e traduzir para o mundo virtual. Isso faz com que o desenvolvedor consiga

Figura 1: Jogo StarWars HoloChess.

colocar objetos virtuais no display sem a necessidade de alvos (tradicionais em aplicativos de AR);

- Visão de cena: o ARKit consegue identificar superfícies ou planos no mundo real, como chão, mesas e paredes. Isso é interessante para se colocar objetos virtuais de acordo com esses planos;

- Rederização: Ele se integra com os frameworks nativos do iOS, como o SpriteKit, SceneKit e o Metal, além de integrar com engines gráficas como a Unity3D.

Alguns conceitos básicos são importantes para o uso do ARKit. São eles:

- ARSession (seção): cada interação onde será usado o AR é uma seção, chamada ARSession. Nela o ARKit irá processar os 60 frames por segundo, pegando informações da câmera e dos sensores. O ARKit possui métodos caso uma seção for interrompida ou se teve alguma falha, de forma que o desenvolvidos possa se adequar caso haja algum erro;

- ARSessionConfiguration: Quando uma ARSession começa ela necessita de uma configuração. Dessa forma, em dispositivos mais poderosos podem usar mais recursos e dispositivos menos poderosos podem usar menos. Também dependendo da sua cena não é necessário o uso de todos os recursos, e alguns podem ser desabilitados economizando bateria do seu usuário (pois ira requisitar menos do dispositivo).

- ARFrame: durante a seção o desenvolvedor consegue pegar as informações do momento, como a imagem capturada da câmera, como esta a cena, o posicionamento, bem como estão as condições de luz da cena;

Para a renderização da cena, é necessário o uso de alguma tecnologia para essa finalidade. Desta forma esta seção abordará as tecnologias que serão usadas para criar um pequeno protótipo com ARKit.

3.1 SpriteKit

SpriteKit é um framework de desenvolvimento de jogos 2D para iOS [6]. Ele foi introduzido no iOS 7 e desde então ele se popularizou bastante. Ele tem suporte para sprites (para quem não sabe, sprite é uma imagem ou animação 2D integrada em uma cena), física, áudio e efeitos visuais, como vídeos, sombras e muito mais coisas legais. Com o iOS 8, ele trás muitos outros features, como shaders e um editor de cena visual.

Sem o SpriteKit, tínhamos de desenvolver nosso próprio framework, usando OpenGL, que é bem mais complexo, ou usar framework externos, como o Cocos2d (que tem um paradigma bem parecido com o SpriteKit) e o Sparrow, ou até mesmo usar engines para fazer o desenvolvimento, como a Unity e a Unreal. Além disto, o SpriteKit tem grandes vantagens, como estar embutido dentro do XCode; ser otimizado para o iOS; possuir suporte para os controles iOS; ter suporte para criação de Atlas, sons e até mesmo cenas (com iOS 8); facilidade de criação de efeitos visuais, como utilização de vídeos dentro de sprites, inclusão de efeitos de luz e sombra ; criação de partículas; e até mesmo integração entre 3D e 2D (com iOS 8 e SceneKit que veremos na próxima seção). A única desvantagem é que o desenvolvimento fica restrito ao ecossistema da Apple, então nada de Android e Windows Phone.

O Framework SpriteKit provê uma estrutura de renderização gráfica para animar texturas, com estrutura geral composta por cenas, onde as cenas são composta por nós (SKNodes). Os nós são modificados a partir de propriedades e métodos, além de funções pré-determinadas (SKAction). Assim como o iOS, o SpriteKit também tem o paradigma de View, só que nesse caso é um SKView, que possui cenas, que são chamadas de SKScenes.

3.2 SceneKit

O SceneKit pode ser pensado como uma extensão do SpriteKit para 3D [6]. Ele tem um paradigma bem parecido com o SpriteKit, e é um framework que é também totalmente integrado com o XCode, e pode até mesmo integrar partes do SpriteKit. O SceneKit foi apresentado inicialmente para OSX em 2012, e agora no iOS 8 ele está disponível para as plataformas moveis da Apple também.

Com o SceneKit é possível renderizar geometria 3D, importar e renderizar modelos COLLADA em suas cenas; manipular materiais, volumes e geometria na cena; fazer load de dados 3D do tipo DAE; usar o XCode como ferramenta de desenvolvimento; criar shaders, luzes e outros efeitos visuais; e usar o asset manager, que converte as geometrias para o padrão do SceneKit, intercala geometrias e comprime texturas para o padrão PVRTC. O SceneKit também possui um editor, onde podemos montar e modificar DAE dentro do próprio XCode, com isso podemos montar cenas sem mexer no código, editando nós, propriedades, hierarquia, atributos e materiais. Dentro do XCode possuímos um outro editor, de partículas, onde podemos criar e editar sistemas de partículas 3D para colocarmos em nossa cena.

O SceneKit fica uma camada abaixo do Cocoa, e uma camada acima do OpenGL (que futuramente deve ser substituído pelo Metal) e é totalmente integrado com o Core Animation e o ImageKit. O SceneKit implementa uma estrutura hierárquica chamada de grafo de cena (mesmo paradigma que o SpriteKit), chamado SCNScene. Mas no caso dessa cena, o nó raiz, que é um SCNNode, define o sistema de coordenado do mundo da cena, e possui filhos que definem o conteúdo da cena. Um exemplo de hierarquia de nós pode ser vista na figura 2.

As animações do SceneKit são baseadas no Framework Core Animation, usando o mesmo poder, sem a necessidade de adição de complexidade, e podem ser animações implícitas ou explicitas. Existem diversas classes no SceneKit que definem propriedade de animações e o desenvolvedor podem criar transições entre as duas animações para ficarem suaves. Para isso temos a classe SCNTransaction, que define como as animações devem se comportar quando na transição entre elas. Para animações mais complexas o desenvolvedor pode fazer uma subclasse da CAAnimation, definindo novos valores de propriedades e comportamentos para a animação. Animações criadas em programas de design 3d também são usadas como CAAnimation.

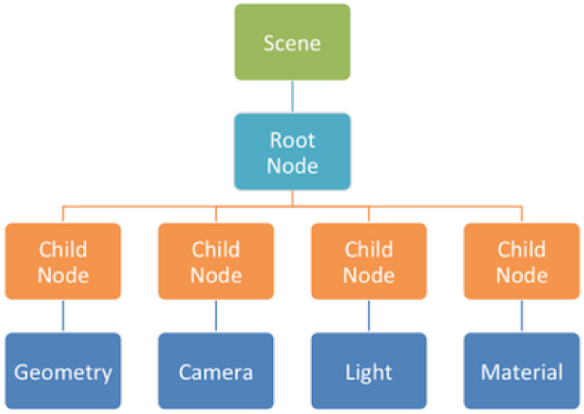

Figura 2: Exemplo de hierarquia de nós no SceneKit.

3.3 Metal

Metal é um novo framework da Apple, construído para usar o poder de processamento dos novos processadores A7, que já esta disponível nos últimos modelos da iOS [6]. Ele pode ser usado otimizar códigos paralelos e games, usando o multiprocessador A7, A8, A9 ou A10 para paralelizar as tarefas. Ele tem um paradigma parecido com linguagens de processamento paralelo em placas gráficas, como CUDA e OpenCL [5], e até mesmo de processadores mobile, como o Renderscript para Android [8]. Mas diferentemente dessas linguagens, ele também permite o processamento das renderizações da cena, podendo ser muito útil para otimizar a fase de render do game, tarefa que normalmente é mais custosa em um game.

Normalmente, para renderizar ou acessar o processamento da placa gráfica, dependemos do OpenGL ES, que traduz as chamadas de funções, para comandos da placa gráfica. Isso causa um overhead que é eliminado com o uso de Metal. Por exemplo, uma simples chamada para renderizar os objetos 3D da cena, pode ser otimizada em até 10x usando o Metal. Além de gráficos, podemos otimizar outras tarefas do game, como por exemplo o processamento de física, uma tarefa que já é otimizada em placas gráficas no PC, como na engine de física Phisyx. Um dos fatores que diferencia essa linguagem para o das placas gráficas, é que existe um compartilhamento de memoria entre a CPU e GPU, fazendo com que o acesso a memoria não seja um bottleneck, como pode ser em casos de CPU-GPU no PC.

O processo normal de usar a GPU pode ser bem custoso, e diversas tarefas são feitas durante o processo de renderização, como validação do estado, compilação de shaders e envio do trabalho para GPU. Na validação de estado, o OpenGL tem que confirmar se o uso da API é valido e encodar esse estado para o hardware. Na compilação de shader, onde o código shader é transformado em código de maquina em run-time, e nas idas e vindas na interação entre o shader e código na CPU. E finalmente no envio de trabalho para GPU, onde os comandos são colocados em batch e se verifica onde a memoria deve ser utilizada. Com o metal, a maioria dessas tarefas que são custosas, são evitadas ou até mesmo eliminadas. Sendo assim os shaders são precompilados em maquina de estados na construção do APP, a validação do estado da API é feito somente no carregamento do conteúdo, deixando somente o uso da GPU para ser feito durante o processo de renderização. O fluxo do pipeline do Metal pode ser visto na Figura 3.

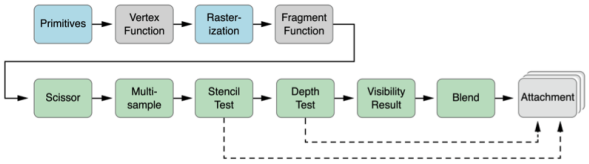

Figura 3: Pipeline do Metal.

3.4 Unity3D

O desenvolvimento de jogos 3D é uma atividade multidisciplinar que envolve diferentes áreas do conhecimento. A complexidade de desenvolver um jogo aumenta a cada dia, necessitando, muitas vezes, de um ano ou mais para o desenvolvimento de um jogo por equipes de aproximadamente cem pessoas. Consequentemente, o uso de ferramentas auxiliares para tarefas repetitivas torna-se fundamental com o objetivo de maximizar o tempo da equipe durante a produção de um jogo. Esse tipo de ferramenta, geralmente conhecido como motor de jogo (game engine), está evoluindo de maneira paralela aos próprios jogos, tornando-se produtos valiosos para o seu desenvolvedor, assim como os próprios jogos.

Com o objetivo de facilitar ao máximo o desenvolvimento de um jogo, alguns módulos e funcionalidades são necessárias para que um motor de jogo seja considerado completo. Inicialmente, o motor de jogo deve possuir um módulo de renderização, com suporte a shaders programáveis via shader languages (HLSL, Cg ou GLSL) ou sistemas próprios de script para shaders. Com o objetivo de possibilitar shaders multiplataformas (OpenGL ou HLSL), alguns plataformas utilizam o seu próprio sistema de linguagem de shader, o qual é posteriormente convertido para a plataforma onde o jogo está sendo executado. Além do módulo de renderização, é necessário também um módulo para programação de scripts que seja relativamente fácil em comparação com linguagens de programação como C/C++. Finalmente, um editor de cenas que permita a importação de assets tais como modelos 3D e áudio, permitindo a sua manipulação na cena de um jogo é imprescindível. Além desses módulos básicos, é desejável também um módulo de física para a detecção e tratamento de colisão e a possibilidade de distribuição do jogo em múltiplas plataformas, sem a necessidade de grande esforço.

A Unity3D é uma ferramenta que inclui o estado da arte no seu segmento, possuindo todos os módulos citados anteriormente, além de módulos adicionais. Como exemplo de módulo adicional, podemos citar um editor de animações integrado à Unity, além de um sistema próprio de controle de versão. É importante ressaltar que nem todos os módulos presentes na Unity são desenvolvidos pela empresa, mas sim módulos independentes que são integrados a ferramenta. Como exemplo, podemos citar o módulo de física, o qual utiliza a PhysX (https://developer.nvidia.com/physx) da nVidia e o módulo de Occlusion Culling, que utiliza o Umbra (http://www.umbrasoftware.com/).

A Unity3D abstrai do desenvolvedor de jogos a manipulação e criação de shaders através da linguagem independente Cg da NVidia. Suporta o desenvolvimento de scripts utilizando as linguagens C#, javascript e Boo. Para execução de scripts em C#, a Unity utiliza uma versão de alto desempenho da biblioteca Mono, sendo uma implementação de código aberto do framework .Net da Microsoft.

Ainda que seja uma ferramenta que inclui todos os módulos necessários para ser considerado um motor de jogo e módulos adicionais, possuindo o estado da arte em seu segmento, a Unity3D possui uma versão grátis disponível, permitindo o desenvolvimento comercial e não comercial de jogos para sistemas Windows e Mac. Além disso, esta versão grátis permite o desenvolvimento de aplicações

voltadas para funcionar em navegadores web e para dispositivos móveis baseadas em iOS, Android e Blackberry, sem nenhum custo adicional. Além desta versão grátis, a Unity3D também possui uma versão comercial, a qual permite a utilização de todas as funcionalidades não disponíveis na versão gratuita, como por exemplo soft shadows, render to target, acesso a API do OpenGL e desenvolvimento de plugins. Adicionalmente, com a aquisição de licenças específicas pode se desenvolver para diversas plataformas, como o Wii-U, Xbox 360 e PS3.

4 ARKIT NO XCODE

Nesta seção vamos criar um projeto com ARKit usando o XCode. Para isso é necessário uma maquina Mac com o XCode 9 instalado. Primeiramente iremos começar com um projeto básico de ARKit que o XCode disponibiliza para a gente. Dessa forma vamos fazer um projeto com elementos do SpriteKit e do SceneKit. Mas para o projeto iremos usar como padrão o projeto básico SpriteKit.

Nele iremos fazer uma aplicação simples, vamos detectar se o ARKit detectou alguma ancora, se sim vamos desenhar um plano em cima da superfície encontrada, e caso esse plano seja clicado, vamos colocar um cubo encima. Dessa forma vamos começar esse aplicativo criando um novo projeto no XCode. No caso vamos selecionar um projeto inicial do ARKit, como mostra a figura 4.

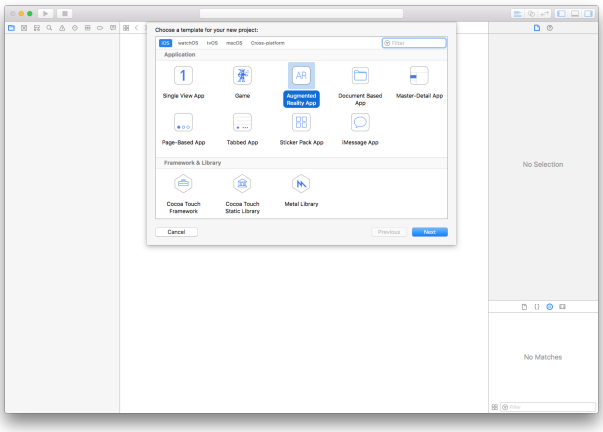

Figura 4: Criando um projeto no XCode.

E clicamos em ”next”, onde iremos fazer as configurações iniciais de como será nosso projeto. Nesta tela iremos colocar o nome que queremos para o projeto, no caso escolhi ”ARApp”, o time de desenvolvedores (isto é, a conta de desenvolvimento necessária para testar no dispositivo e futuramente colocar na App Store), a linguagem que iremos usar, no caso Swift, e a tecnologia que usaremos para renderizar os objetos, que no caso será a SceneKit, como mostra na figura 5.

E selecionamos o local onde queremos salvar nosso projeto, no caso estou selecionando a pasta de ”Downloads”, como mostra a figura 6.

Se rodarmos este projeto, com um dispositivo, já que em um simulador não funciona o ARKit, pois não tem câmera. Nele veremos a câmera com uma nave colocada no centro da tela, como mostra a figura 7.

Iremos modificar somente o arquivo ViewController.swift, onde terá toda a nossa lógica do nosso aplicativo. Nele, depois dos import, podemos ver no código abaixo que temos a definição da classe ViewController, que extende um UIViewController, classe que realiza as funções basica de uma tela de aplicativo e implementa o ARSCNViewDelegate, que seria as principais funções do ARKit.

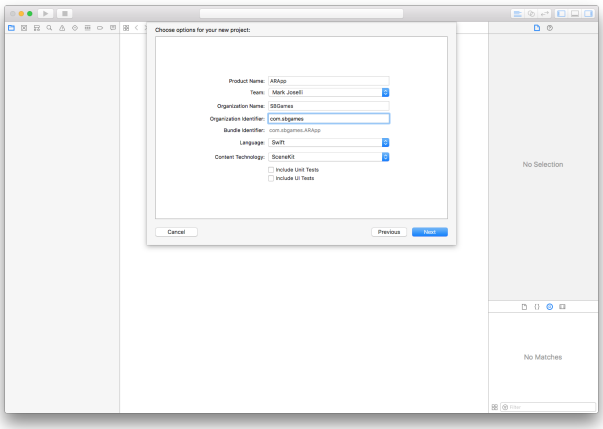

Figura 5: Configurações iniciais do projeto.

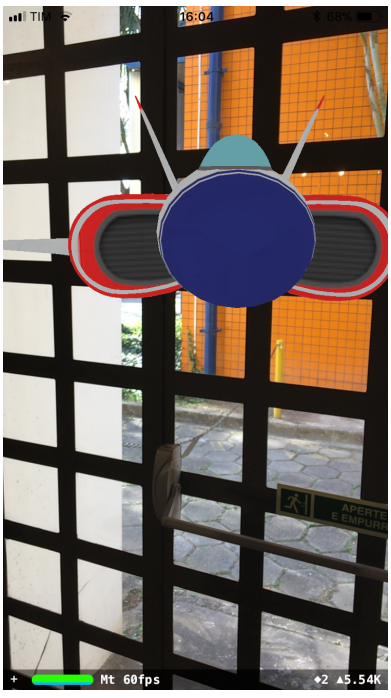

Figura 6: Selecionando o local do projeto.

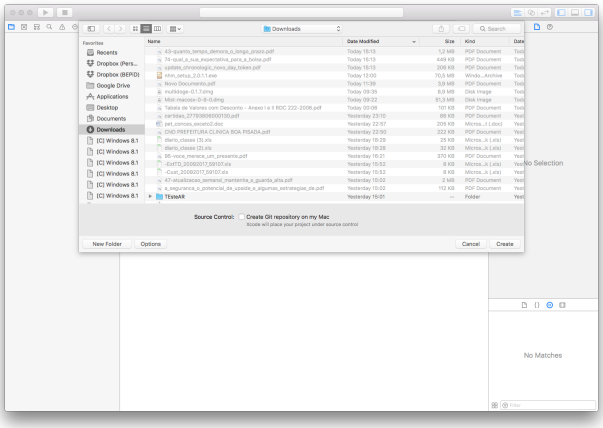

Figura 7: Rodando o projeto base.

E logo depois temos a SceneView, que é uma ARSCNView, uma cena do sceneKit que tem os recursos necessários para renderizar as cenas juntamente com as capturas de câmera e outras característica do ARKit.

4.1 Configurando a cena

Ai vem a definição do ViewDidLoad, função que é chamada logo o tela ser criada, de forma que qualquer carregamento e configuração deve ser feita nesta função. Desta forma vamos modificar um pouco esta função de forma a ter uma cena vazia, sem essa nave flutuando no espaço e mostrar os feture points, que são pontos que o ARKit detecta características na cena. O código desta função pode ser vista abaixo.

Agora iremos para a ViewWillAppear, função que é chamada antes de mostrar para o usuário que a tela. Nela temos de criar a seção de AR que iremos usar. No caso iremos inicializar ele com uma configuração padrão de ARWorldTrackingConfiguration e fazendo com que o ARKit detecte planos no horizontal. Atualmente somente existe esta possibilidade de detecção, mas abre a possibilidade do framework permitir outros tipos no futuro. E vamos configurar a ViewWillDisappear, que irá pausar a seção, para caso não deixar o ARKit processando caso a tela não mais ser mostrada para o usuário. Estas funções pode ser vista abaixo.

Rodando o código ainda não vemos objetos em cena, mas vemos os features points, como mostra a Figura 8.

4.2 Criando objetos de acordo com as ancoras

Agora que temos uma ARSession, podemos usar o SceneKit para colocar objetos virtuais interagindo com os objetos reais capturados pela câmera. Quando a detecção de planos esta habilitada, como fizemos no viewWillAppear, o ARKit adiciona e atualiza ARAnchor (ou ancoras). Dessa forma, assim que o ARKit detectar um novo plano, vamos desenhar este plano na cena. Para isso usa-se a as coordenadas do ARArchor, transformando-o em um ARPlaneArchor. Ai segue-se o processo do SceneKit, onde primeiramente se cria a geometria do objeto, no caso o SCNPlane, e depois se adiciona essa geometria em um SCNNode. Depois disso se posiciona o plano e rotaciona ele (pois ele nativamente esta perpendicular ao eixo y), damos um pequeno alpha, modificando a opacidade e adicionamos ao nó do ARKit, como mostra o código abaixo.

Apos adicionar este plano, devemos atualiza-lo ou ajusta-lo, conforme as condições mudam, dessa forma, iremos usar o did update para colocar este plano com o tamanho e a posição conforme o ARAnchor, como mostra o código abaixo.

Figura 8: Rodando o projeto com e vendo feature points.

Com isso podemos testar o nosso aplicativo e podemos ver que ele consegue detectar planos, como mostra a figura 9.

4.3 Interagindo com a cena

Agora vamos colocar que se o usuário dar um tap na tela, ele irá criar um cubo caso o raycast (traçar um raio partindo do ponto pressionado) encontre algum plano do ARKit. Dessa forma vamos adicionar um código no viewDidLoad, para criar um reconhecedor de tap, que quando o gesto for reconhecido ele irá chamar a função tapped, como mostra o codigo abaixo.

A função tapped irá pegar a localização do toque, e verificar se o raycast irá atingir algum plano do ARKit. Caso ele atinga, ele pegará o primeiro e irá colocar um cubo de acordo com o local onde teve este toque, criando primeiramente a geometria e depois o nó, como mostra o código abaixo.

Dessa forma, quando houver um plano e o usuário dar um tap na direção deste plano, ele irá adicionar uma box, como mostra a figura 10.

5 ARKit na Unity

Para termos o ARKit na Unity, precisamos da Unity no mínimo na versão 5.6.2 e os pacotes para suporte iOS (de forma que exportemos o projeto). Também precisamos do Unity ARKit Plugin, que pode ser baixado gratuitamente em https://www.assetstore.unity3d.com/en/#!/content/92515, ou no Asset Store, como na figura 11.

Tendo feito o download podemos criar um novo projeto, como mostra a figura 12.

Adicionando em seguida o ARKit (clicando no assets catalog), como mostra a figura 13.

Tendo feito isso, com o projeto já criado, podemos abrir uma das cenas que vem no package, a ?UnityARKitScene?. que já coloca em cena os principais componentes que vamos precisar. Como o ARKit é exclusivo do iOS, precisamos mudar a plataforma para build, clicando no menu ”File-¿Build Settings”, selecionando iOS

Figura 9: Rodando o projeto e vendo a detecção de planos.

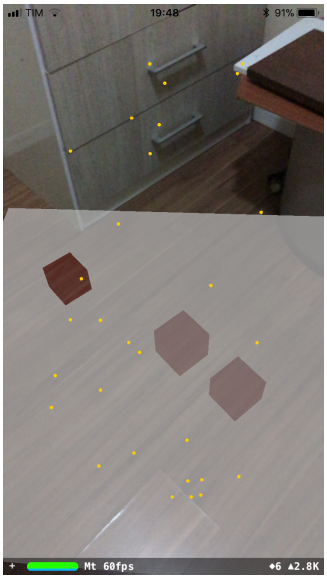

Figura 10: Rodando o projeto e criando cubos.

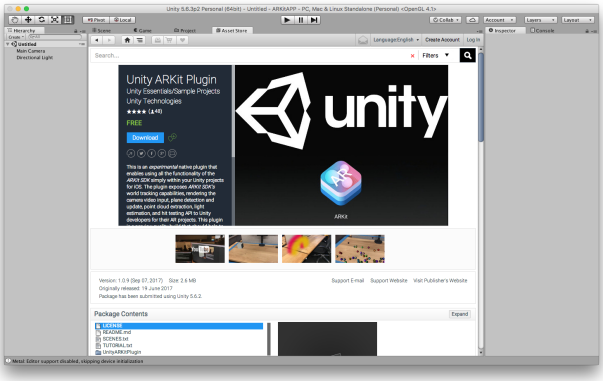

Figura 11: ARKit Plugin na Asset Store.

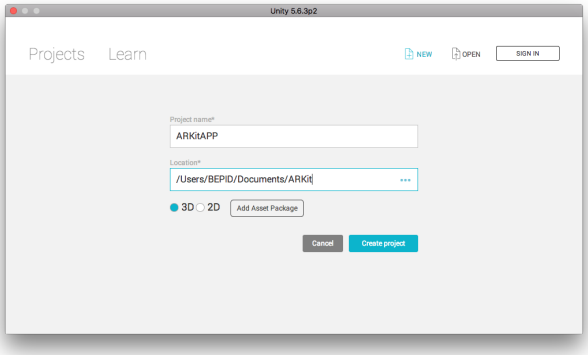

Figura 12: Criando um projeto na Unity.

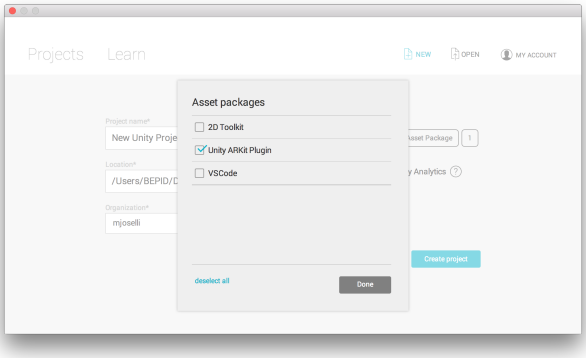

Figura 13: Adicionando o plugin ao projeto.

e clicando em ”Switch Platform, como mostra a figura. No caso se queira testar, deve-se clicar em ”Build and Run”, onde a Unity irá gerar um projeto do XCode e mandar rodar, como mostra a figura 14.

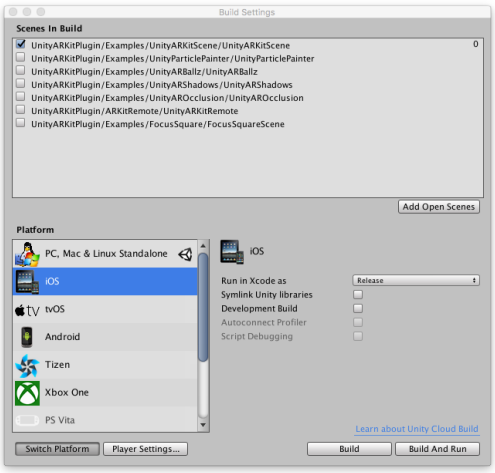

Figura 14: Mudando a plataforma da Unity.

A Unity irá fazer o import ajustando para a plataforma selecionada e termos nossa cena pronta para mexermos, como mostra a figura 15.

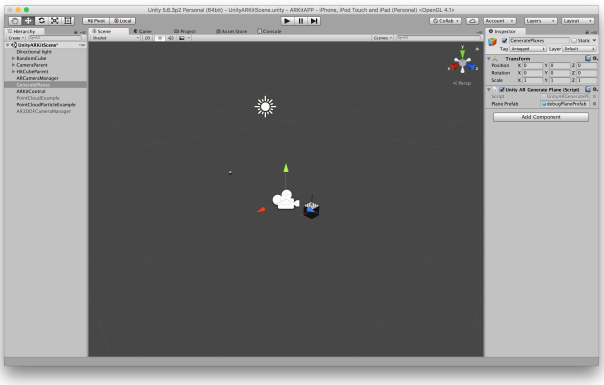

Figura 15: A cena usada no exemplo.

Nesta cena, podemos ver os principais funcionalidades do ARKit agrupado em componentes da Unity. Para controlar a câmera, e fazer ela se comportar com a câmera do celular e com os sensores tem o Unity AR Camera Manager, que tem como referencia a câmera principal da cena, e consegue customizar alguma das opções do ARKit, como se queremos que detecte planos, se deve ter um calculo de luz estimado, se deve pegar pontos de features e como deve ser o alinhamento. Este componente já prepara todo o ambiente para o ARKit, criando a configuração e rodando o ARSession. O inspector desse componente pode ser visto na Figura 16.

Para realizar a tarefa de detectar archor e colocar os planos, o plugin disponibiliza o script ”UnityARGeneratePlanes”, que realizar praticamente a mesma tarefa que fizemos no código em iOS. Este plugin somente precisa de referenciar um prefab que será utilizado quando um ARArchor for achado. O inspector deste componente pode ser visto na figura 17.

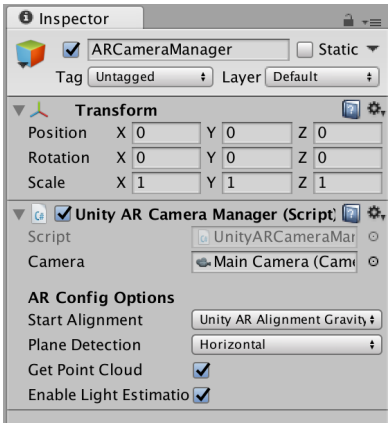

Figura 16: Inspector da Camera do ARKit.

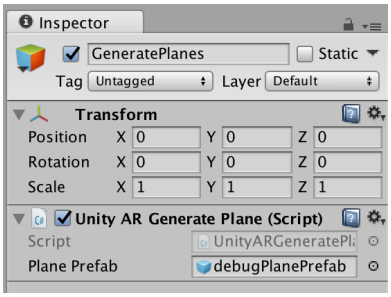

Figura 17: Inspector do UnityARGeneratePlanes.

No script ele tem um UnityARAnchorManager que faz todo o processo de adicionar e remover ARKitArchor internamente. Ai fica para a UnityARGeneratePlanes fazer algo com os Archors verificando durante o loop onde eles estão e como podem ser usados.

Agora para verificar colisões a cena tem o script UnityARHitTestExample dentro do HitCube, onde caso ele tenha um toque na tela (ou movimentação) ele transforma o objeto referenciado em HitTransform de acordo com o resultado do hei test entre o raycast do toque e o plano gerado pelo ARKit. O inspector deste componente pode ser visto na figura 18.

Neste script ele realiza o mesmo processo de durante o loop pegar o input, transformar para as coordenadas do viewport, e verificar se houve algum plano esta na diração do raycast lançado do ponto do toque, rodando a função HitTestWithResultType para cada tipo de ARHitTestResultType, que são os objetos que ele pode encontrar durante o raycast, que podem ser planos estendidos, planos que existem, planos horizontais ou até mesmo pontos de features como mostra o código abaixo.

Para realizar a função HitTestWithResultType é necessario usar o HitTest usando a seção do ARKit está executando. Caso exista um hit, o código irá reposicionar o cubo (ou outro GameObjet referenciado) de acordo com os resutados desse hit, como mostra o çódigo abaixo.

Com isso podemos testar o nosso aplicativo e podemos ver que conseguimos posicionar o cubo de acordo com a cena AR, como mostra a figura 19.

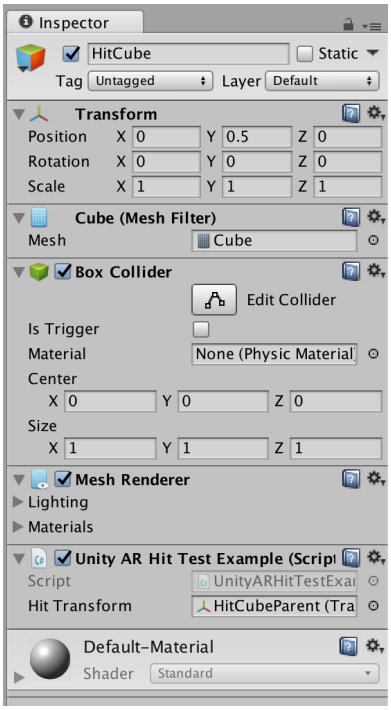

Figura 18: Inspector do UnityARHitTestExample.

Figura 19: Criando um projeto no XCode.

6 Conclusões

Aplicações em AR estão ficando cada vez mais comuns, mas suas possibilidades de uso ainda estão sendo descobertas. Este capitulo não pretende cobrir por completo todas as funcionalidades que podem ser exploradas em aplicações em AR. Porém, pretende fornecer ao leitor a base necessária para construir suas próprias aplicações de realidade aumentada a partir dos conceitos demonstrados. Além disso, pretende-se também fomentar no leitor o desejo pela busca de novas informações e possibilidades que podem ser desenvolvidas utilizando AR usando a facilidade do ARKit.

Referências

[1] J. R. da Silva Junior, M. Joselli, E. Clua, M. Pelegrino, and E. Mendonça. An architecture for new ways of game user interaction using mobile devices. In *SBGames*. SBC, 2011.

[2] O. Faugeras. *Three-dimensional Computer Vision: A Geometric Viewpoint*. MIT Press, Cambridge, MA, USA, 1993.

[3] R. Fisher, K. Dawson-Howe, A. Fitgibbon, C. Robertson, and E. Trucco. *Dictionary of Computer Vision and Image Processing*. Wiley, 2005.

[4] M. Joselli and E. Clua. grmobile: A framework for touch and accelerometer gesture recognition for mobile games. In *2009 VIII Brazilian Symposium on Games and Digital Entertainment*, pages 141–150, Oct 2009.

[5] M. Joselli, J. R. da Silva Junior, , E. Clua, and E. Soluri. Mobilewars: A mobile gpgpu game. *Lecture Notes in Computer Science*, 8215, 2013.

[6] M. Joselli and M. Fernandes. Desenvolvimento de jogos para plataforma ios - presente e futuro. In *SBGames 2014 - Tutoriais*, Porto Alegre, RS, 2014.

[7] M. Joselli, J. S. Junior, and E. Clua. Realidade aumentada em games com uso da unity3d. In *SBGames 2013 - Tutoriais*, São Paulo, SP, oct 2013.

[8] M. Joselli, Passos, J. S. Junior, M. Zamith, E. Clua, and E. Soluri. A flocking boids simulation and optimization structure for mo-bile multicore architectures. In *XI Brazilian Symposium on Computer Games and Digital Entertainment, SBGames 2012 ? Computing Track*, 2012.

[9] R. Koster. *Theory of Fun for Game Design*. O'Reilly Media, Inc., 2nd edition, 2013.