Alexandre Pizarro Drummond Abrahão Ferreira

Desenvolvimento de um jogo multijogador usando Realidade Aumentada para simular um carro de controle remoto

RELATÓRIO DE PROJETO FINAL

DEPARTAMENTO DE ENGENHARIA ELÉTRICA E DEPARTAMENTO DE INFORMÁTICA

Programa de graduação em Engenharia de Computação

Rio de Janeiro
janeiro de 2022

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Alexandre Pizarro Drummond Abrahão Ferreira

Desenvolvimento de um jogo multijogador usando Realidade Aumentada para simular um carro de controle remoto

Relatório de Projeto Final

Relatório de Projeto Final, apresentado ao programa de Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do titulo de Engenheiro de Computação.

Orientador: Prof. Alberto Barbosa Raposo

Rio de Janeiro
janeiro de 2022

Resumo

Abrahão, Alexandre; Raposo, Alberto. **Desenvolvimento de um jogo multijogador usando Realidade Aumentada para simular um carro de controle remoto**. Rio de Janeiro, 2022. 39p. Projeto de Graduação – Departamento de Engenharia Elétrica e Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A relativa democratização de *smartphones* com hardwares poderosos vem possibilitando que jogos mobile possam trazer novas experiências. Portanto, o objetivo deste projeto é experimentar possibilidades de controlar um carro de controle remoto, usando Realidade Aumentada. Para isso, foi desenvolvido um protótipo de um jogo multijogador para *iOS* para simular um carro de controle remoto usando Realidade Aumentada, com o intuito de ser publicado *AppStore*.

Palavras-chave

Realidade Aumentada; jogo; multiplayer; mobile; iOS; Swift

Sumário

1 Introdução **5**
1.1 Motivação 5
1.2 Definição do problema 6

2 Situação Atual **8**
2.1 Swift 9
2.1.1 UIKit 9
2.1.2 ARKit 9
2.1.3 SceneKit 9
2.1.4 SpriteKit 9
2.1.5 AVFoundation 10
2.1.6 MultipeerConnectivity 10
2.1.7 GameplayKit 10
2.1.8 simd 10
2.2 Softwares 10
2.2.1 Xcode 10
2.2.2 GarageBand 11
2.2.3 MagicaVoxel 11
2.2.4 Blender 12
2.2.5 Sketch 12
2.3 Soluções similares e limitações 12
2.3.1 Limitações de hardware 14
2.3.2 Limitações de software 15

3 Proposta e objetivos **16**

4 Atividades Realizadas **17**
4.1 Processo de descoberta e pesquisa 17
4.1.1 CBL 17
4.1.2 Formulário de pesquisa 19
4.1.3 Definição da funcionalidade diferencial 20
4.2 Estudos da tecnologia disponível 21
4.2.1 AR Multiuser 21
4.2.2 SwiftShot 22
4.3 Fluxos de trabalho 23

5 Projeto e especificação do sistema **24**
5.1 Arquitetura e design de projeto 25
5.1.1 Model-View-Controller (MVC) 25
5.1.2 Entity-Component System (ECS) 26
5.1.3 Networking 28
5.2 Inversão de dependência em Swift 28
5.3 Game design 30

6 Implementação e avaliação **32**
6.1 Compartilhamento de mundo 32
6.2 Compartilhamento da física 33
6.3 Problemas com a simulação veicular 34
6.4 Testes 35
6.4.1 Testes em celulares diversos 36
6.4.2 Testes de interface gráfica e funcionalidades do jogo 36
6.4.3 Testes unitários automatizados 37

7 Considerações finais **38**

Referências bibliográficas **39**

1
Introdução

1.1
Motivação

O mercado de smartphones e a capacidade de processamento dos dispositivos móveis vêm crescendo significativamente nos últimos anos. O número de pessoas que carregam um hardware potente no bolso continua a aumentar, o que encoraja desenvolvedores de software a encontrar novas maneiras de integrar tecnologias interessantes no cotidiano. Assistentes de voz, filtros de vídeos e muitas outras aplicações que impactam na vida de centenas de milhares de pessoas foram possibilitadas pela popularidade de smartphones avançados.

Uma das principais tecnologias que vem sendo popularizada pelo smartphone é a Realidade Aumentada (*AR*, do inglês *Augmented Reality*). Ela pode ser definida por três características principais: combina o real com o virtual, é interativa em tempo real e é registrada em três dimensões (Azuma 1997). No geral, *AR* envolve processamento de imagens de uma câmera para adicionar algum efeito novo no ambiente capturado.

Alguns exemplos de usos populares de *AR* são encontrados nos aplicativos *Instagram*, o qual possibilita aplicar filtros animados em fotos e vídeos, e da fabricante de móveis *IKEA*, que permite posicionar seus móveis virtualmente na casa do cliente.

Ao longo da graduação, tive a chance de desenvolver algumas aplicações conceituais para *iOS* em *AR* e realizar estudos na área de Visão Computacional sobre os fundamentos de processamento de imagens para adicionar conteúdo em três dimensões com perspectiva correta. Isso evidenciou o potencial da tecnologia, especialmente no âmbito de jogos virtuais.

Nesse contexto, percebi a possibilidade de unir a tecnologia de *AR* com uma paixão pessoal sobre o automobilismo. Portanto, comecei a desenvolver um jogo de controlar um carro em *AR* para a plataforma *iOS* como entrada de um desafio estudantil. Então, avaliei ser interessante explorar o potencial desse jogo neste trabalho, que tem como objetivo aprimorá-lo para a publicação na loja de aplicativos.

1.2 Definição do problema

Ao pesquisar por jogos que utilizam *AR* na *AppStore*, a loja virtual de aplicativos da plataforma *iOS*, é possível ver que muitos utilizam a tecnologia de maneira similar. É comum ver funcionalidades que o jogador irá achar intrigante em um primeiro momento, mas acabará usando pouco. Um exemplo bem frequente disso é posicionamento de objetos virtuais do jogo com apenas interações simples (rotacionar e ampliar).

Limitando a busca por jogos de corrida, a funcionalidade de *AR* normalmente é usada para colocar os carros do jogo no ambiente, com algumas opções de customização visual. É o caso nos jogos *Need For Speed™ Heat Studio* (EA 2019) e *CSR Racing 2* (NaturalMotion 2015), mostradas nas figuras 1.1 e 1.2, respectivamente.

Figura 1.1: Carro em AR no jogo Need For Speed™ Heat Studio.

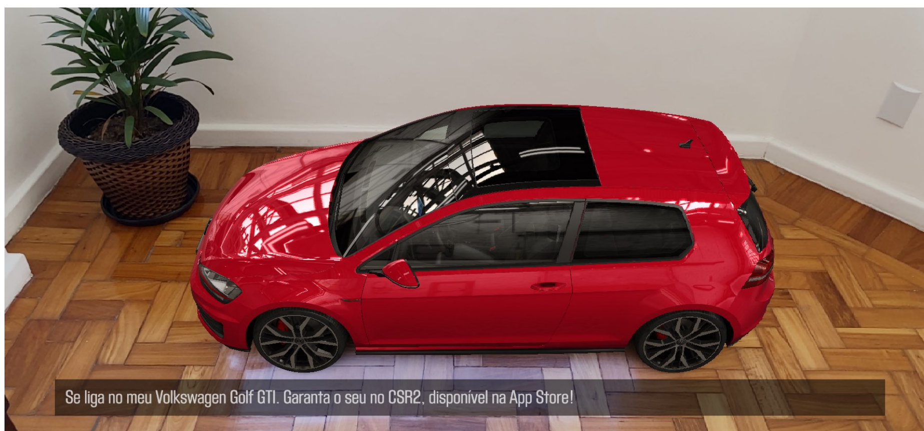


Figura 1.2: Carro em *AR* no jogo *CSR Racing 2*.

Além disso, em geral, os jogos mobile em sua grande maioria são recheados de compras dentro do aplicativo, como de moedas virtuais e pacotes de

melhorias contendo conteúdos aleatórios, chamados de *"Loot Boxes"*, os quais incentivam o comportamento similar a jogos de azar (Zendle, Cairns 2018). A progressão e o conteúdo desses jogos são frequentemente bloqueados atrás de barreiras monetárias ou precisam de um tempo de jogo bem elevado para serem desbloqueados.

Este projeto propõe estudar uma forma mais interessante de usar *AR* para promover experiências engajadoras e memoráveis em um jogo de carro de controle remoto.

2
Situação Atual

O jogo proposto, nomeado de *Synth Car*, foi projetado inicialmente como uma entrada para um concurso de estudantes a fim de ganhar um ingresso para a *Worldwide Developers Conference* de 2019 (*WWDC*), conferência anual na qual os desenvolvedores para plataformas *Apple* (*iOS*, *MacOS*, etc) se reúnem para trocar conhecimento e saber em primeira mão das novidades (Apple 2021). Portanto, foi desenvolvido para a plataforma *iOS*, sistema operacional do smartphone da *Apple*, o *iPhone*.

O jogo possui visuais inspirados na estética futurista dos anos 80, com efeitos de iluminação e cores vibrantes contrastando com escuro, como pode ser observado na figura 2.1.

Figura 2.1: Cena do jogo *Synth Car*.

2.1 Swift

Criada em 2014, a linguagem de programação *Swift* foi desenvolvida pela *Apple* para ser fácil de ser aprender, reduzir erros comuns de programação e ter uma boa performance. Além disso, ela veio para se tornar a linguagem nativa para desenvolvimento de aplicações para as plataformas *Apple*, substituindo a antiga linguagem *Objective-C* nessa tarefa (TNW 2014). O jogo foi escrito integralmente em *Swift*, o que era um requisito do concurso da *WWDC*.

A *Apple* disponibiliza em *Swift* várias bibliotecas nativas para acelerar o desenvolvimento de componentes comuns entre aplicativos, como construção de interfaces, manipulação de áudio e vídeo, entre outras. As bibliotecas em foco neste projeto serão: *UIKit*, *ARKit*, *SceneKit*, *SpriteKit* e *AVFoundation*, as quais foram utilizadas na versão inicial do jogo, além de *MultipeerConnectivity*, *GameplayKit* e *simd*, que foram adicionadas ao jogo no decorrer do projeto.

2.1.1 UIKit

É uma biblioteca fundamental para o desenvolvimento de aplicativos, pois contém a base para todo conteúdo visual da interface. Além disso, fornece maneiras de navegar entre telas e gerenciar eventos de interação do usuário.

2.1.2 ARKit

É uma biblioteca que reúne ferramentas para agilizar o desenvolvimento de experiências em *AR* e melhorar a performance das operações complexas requeridas por tais experiências.

2.1.3 SceneKit

É uma *API* para simplificar o desenvolvimento de conteúdo em 3D, sem precisar conhecer sobre outras *APIs* de mais baixo nível. O *ARKit* usa o *SceneKit* para renderizar os conteúdos em 3D.

2.1.4 SpriteKit

É uma *API* para simplificar o desenvolvimento de jogos 2D. No contexto desse projeto, ela pode ser usada para renderizar a interface do jogo, como os controles de movimento do carro.

2.1.5 AVFoundation

É uma *API* para simplificar o gerenciamento de conteúdo audiovisual, como músicas e vídeos. Possibilita adicionar músicas e fundo e efeitos sonoros ao jogo.

2.1.6 MultipeerConnectivity

É uma *API* que simplifica a interconexão entre dispositivos de plataformas *Apple*. Ela gerencia o meio de transmissão de dados, escolhendo automaticamente entre *Wi-Fi*, *Bluetooth* ou *Ethernet*, conforme a disponibilidade.

2.1.7 GameplayKit

É uma *API* desenvolvida para auxiliar desenvolvedores a arquitetar e organizar a lógica de um jogo.

2.1.8 simd

SIMD é uma sigla que significa *Single Instruction Multiple Data*. É um módulo que provê operações vetoriais e matriciais otimizadas, usadas pelo *SceneKit* para renderizar as cenas e manipular os objetos 3D.

2.2 Softwares

Para o desenvolvimento do projeto, múltiplos softwares foram estudados e utilizados durante o processo para suprir as necessidades específicas de cada etapa.

2.2.1 Xcode

Para trabalhar em projetos de aplicativos *iOS*, a *Apple* disponibiliza a ferramenta chamada de *Xcode*, que é um ambiente de desenvolvimento integrado (sigla em inglês *IDE*). Com ele, é possível compilar programas escritos em *Swift* e executar em simuladores das plataformas *Apple*, como *iPhone* ou *Apple Watch*, em dispositivos reais, ou mesmo em linha de comando. No jogo deste trabalho, todos o arquivos de código-fonte e recursos audiovisuais precisam estar em um projeto criado pelo *Xcode* para serem incluídos no aplicativo e, consequentemente, instalados no smartphone.

É importante ressaltar que as versões do *Xcode* são feitas para executar somente em computadores que executam o sistema operacional *macOS*, ou seja, somente computadores fabricados pela *Apple*.

2.2.2 GarageBand

Os recursos sonoros do jogo, como música de fundo e efeitos sonoros, precisam ser criados utilizando um software de edição e manipulação de áudio. Para isso, existe o *GarageBand*, que é o software também criado pela *Apple* para tal função. Como o desenvolvimento deste projeto foi feito utilizando um *macOS*, foi possível usá-lo.

2.2.3 MagicaVoxel

Os modelos 3D do jogo, como os carros e os cenários, foram feitos usando o software gratuito chamado *MagicaVoxel*. Esse programa permite modelar objetos rapidamente usando *voxels*, que, na computação gráfica, são elementos discretos que compõem uma entidade 3D (Merriam-Webster 2021). No contexto da ferramenta, cada *voxel* é uma unidade cúbica de construção, que produz modelos como da figuras 2.2 e 2.3.

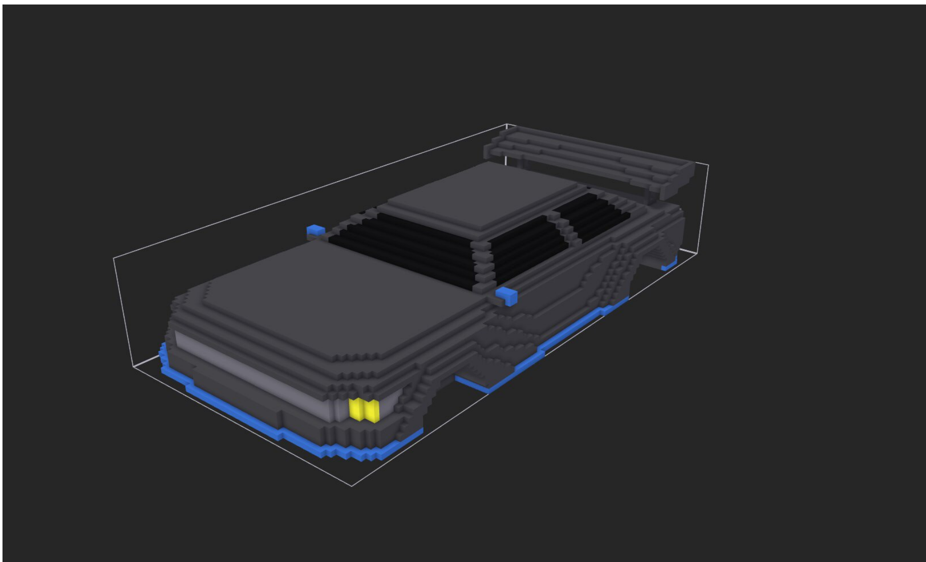


Figura 2.2: Frente do modelo 3D do chassis do carro, no software *MagicaVoxel*.

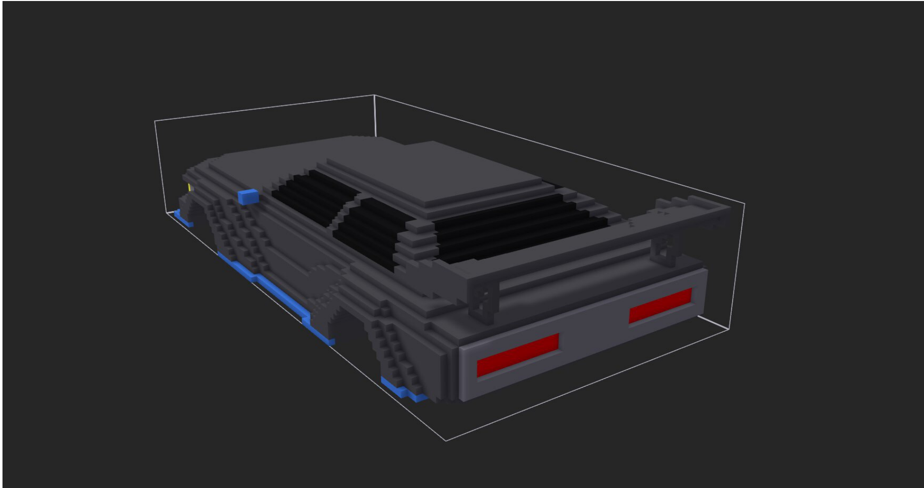


Figura 2.3: Traseira do modelo 3D do chassis do carro, no software *MagicaVoxel*.

2.2.4 Blender

Os modelos 3D gerados pelo *MagicaVoxel* são exportados em formatos que não funcionam bem ao serem importados pelo projeto no *Xcode*. Além disso, a escala dos objetos 3D precisa ser ajustada para eles terem dimensões corretas entre si, o que também não é suportado pelo *MagicaVoxel*.

Para resolver esse problema, é usado o *Blender*, que também é um software gratuito de modelagem 3D, feito para trabalhar com os mais diversos tipos de modelo. No contexto deste projeto, o *Blender* foi usado para executar o passo intermediário entre a exportação de modelos do *MagicaVoxel* e a importação destes pelo *Xcode*.

2.2.5 Sketch

Para criar os elementos da interface do jogo na tela do celular, é preciso de um software que possa criar recursos de imagem 2D. Portanto, foi escolhido o software *Sketch* para tal, já que possuo uma licença de desenvolvimento e seus recursos atendiam bem às necessidades de criação.

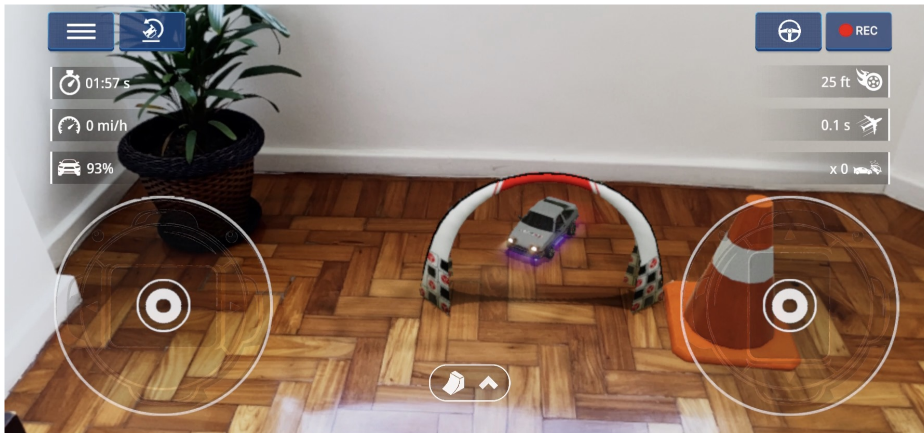
2.3 Soluções similares e limitações

Além dos jogos apresentados na seção 1.2, existem outros dois jogos disponíveis na *AppStore* que vão um passo além de apenas posicionar carros

virtuais 3D em um ambiente real. Eles permitem controlar tais carros, similar ao proposto neste projeto.

O primeiro e mais popular se chama *RC Club*. Lançado em 2018, possui uma variedade de carros para controlar e modos de jogo que permitem criar pistas de corrida e posicionar . Em sua última atualização, recebeu suporte ao sensor *LiDAR*, presente em alguns *iPads* e *iPhones* mais novos (Abylight 2020). Esse sensor mapeia em 3D o ambiente e fornece informações sobre o terreno que são usadas pelo jogo para informar sobre obstáculos e inclinação. Isso aprimora o realismo das interações, pois o carro virtual consegue interagir com mais fidelidade com objetos reais.

Porém, ele sofre com uma baixa taxa de quadros por segundo (*FPS*, do inglês *Frames Per Second*) e renderiza os *assets* em qualidade inferior aos demais jogos, além de ter sistemas de progressão do jogador que contém conteúdo desbloqueado por *"Loot Boxes"*.



.

Figura 2.4: Jogo *RC Club*, com objetos virtuais posicionados.

Além dele, existe um outro jogo mais simples chamado *AR Race Car*. O jogo é inteiramente grátis, contendo 6 carros para controlar. Porém, sua simulação física é problemática, pois manter o controle do carro é bem difícil.

Figura 2.5: Carro do jogo *AR Race Car*.

2.3.1 Limitações de hardware

Como visto nos exemplos de soluções similares, experiências em *AR* demandam bastante do hardware, sendo bem comum gastar muita energia, drenando a bateria do celular rapidamente, além de esquentar o dispositivo. Por conta disso, os jogos costumam reduzir a qualidade dos *assets* e/ou reduzir os *FPS*. Em particular, a redução de *FPS* afeta bem negativamente na experiência, pois reduz a fluidez das interações do jogador.

Além disso, a simulação física das interações de um carro também impacta o desempenho e gasto de energia, o que contribui para o ponto discutido acima.

Para posicionar corretamente os objetos na cena em *AR*, é preciso determinar informações sobre o ambiente a partir das imagens da câmera. Isso faz com que uma boa iluminação do ambiente seja imprescindível. Superfícies escaneadas com texturas bem definidas, como mesas de madeira, também ajudam na detecção do ambiente, pois facilitam o posicionamento de *feature points* na cena, os quais são essenciais no posicionamento da cena em relação ao mundo real.

Por fim, como o *ARKit* exige muito poder de processamento, a *Apple* limita o uso dele em dispositivos que possuem processadores *A9* ou mais recentes[1], ou seja, *iPhones* e *iPads* lançados a partir de setembro de 2015. Para impedir a distribuição do aplicativo em celulares que não o suportam, a *Apple* disponibiliza um atributo de configuração do projeto, que automaticamente faz essa verificação ao lançar o aplicativo na *AppStore*.

[1] `https://developer.apple.com/documentation/arkit/verifying_device_support_and_user_permission`

2.3.2 Limitações de software

Além das limitações de hardware, para trabalhar com *ARKit* é preciso ter um *iPhone* ou *iPad* rodando pelo menos a versão *iOS* 11.0 do sistema operacional. Como é necessário o uso da câmera do dispositivo, o aplicativo precisa pedir permissão ao usuário para usar tal recurso, por questões de privacidade.

3
Proposta e objetivos

O objetivo deste trabalho consiste em aprimorar o jogo *Synth Car* para propiciar a melhor experiência possível de controlar um carro de controle remoto, dentro dos limites de tempo do projeto. Portanto, foi preciso transformá-lo em um aplicativo que possa ser distribuído no *TestFlight*, que é a plataforma de distribuição de versões beta da *AppStore*.

Para a entrega de um jogo que atende ao descrito acima, foram estabelecidos os seguintes objetivos:

– Definir um público-alvo de jogadores
– Definir regras e funcionalidades para o jogo que sejam diferenciais
– Projetar os níveis e objetivos do jogador
– Criar modelos 3D e imagens 2D baseados no estilo visual retrô futurista para compor os *assets* visuais do jogo, utilizando os softwares mencionados no capítulo 2
– Gravar efeitos sonoros e música temática ou selecionar disponíveis em lojas de conteúdo sonoro
– Definir um modelo de monetização, ou decidir por publicar o jogo gratuitamente
– Estudar sobre as bibliotecas mencionadas na seção 2.1
– Definir requisitos mínimos de sistema e hardware para rodar o jogo, baseado em pesquisa sobre a base de usuários das plataformas *Apple* e versões das *APIs* utilizadas
– Implementar as regras, física dos carros e funcionalidades do jogo
– Criar uma conta de desenvolvedor na plataforma *Apple Developer* para poder utilizar ferramentas de distribuição de versões de teste e publicação na *AppStore*
– Coletar feedbacks, testando iterativamente versões do jogo com o público-alvo

4
Atividades Realizadas

O desenvolvimento do trabalho pode ser dividido em duas etapas principais, que correspondem às disciplinas de Projeto Final I e II, respectivamente. A primeira foi a etapa de descobrimento sobre os requisitos do projeto e os diferenciais que o destacariam diante a competição, além do estudo de tecnologias potenciais a serem utilizadas. A segunda, mais complexa, foi a de elaborar fluxos de trabalho para implementar o jogo a partir das descobertas e dos requisitos que foram definidos na etapa anterior.

4.1
Processo de descoberta e pesquisa

Para guiar o processo de pesquisa sobre design e requisitos do jogo, utilizei um framework chamado *Challenge Based Learning*, ou CBL (The Challenge Institute 2018). Para tirar algumas dúvidas sobre público-alvo foi elaborado um formulário de perguntas e respostas dinâmicas.

4.1.1
CBL

O framework de aprendizado CBL foi elaborado a partir do projeto *"Apple Classrooms of Tomorrow—Today"* (ACOT^2) (Apple 2008), o qual teve como objetivo identificar, junto a educadores, os princípios essenciais do de ambientes de aprendizagem do século XXI.

O CBL, como o próprio nome indica, é baseado na elaboração de um desafio para guiar a pesquisa e o desenvolvimento. É definido por um ciclo que contém três fases, na ordem: Engajar, Investigar e Agir, como mostrado na figura 4.1.

Figura 4.1: Fases do CBL (The Challenge Institute 2018).

A fase de engajamento é o momento em que é definida uma temática central, ampla, para direcionar a pesquisa. É a chamada “grande ideia” (do inglês, “Big Idea”), que escolhi como “Carro de controle remoto em AR”. A partir dela, elaborei uma pergunta principal para refletir as dúvidas sobre o assunto e começar a pesquisa, chamada de “Essential Question”. Tal pergunta inicialmente foi: “Como promover uma experiência divertida de carros de controle remoto em AR?”. Por fim, é definido um desafio inicial, que é uma chamada para ação, definido no CBL como “Challenge”. Minha definição de frase de desafio foi: “Melhorar o projeto Synth Car para se tornar um jogo completo de carro de controle remoto em AR”.

A segunda fase é a de investigação. Nela, a partir da Essential Question, fui elaborando perguntas sobre o assunto (chamadas de “Guiding Questions”), das mais gerais às mais específicas. Tais perguntas foram classificadas em duas categorias principais: “Design e Conceitos” e “Tecnologia”, e foram armazenadas em uma planilha para serem respondidas ao longo do processo e servirem de documentação.

Por fim, a fase de ação é o momento de desenvolver a solução a partir da investigação feita. Essa fase será melhor detalhada na seção 4.2.

Também, é importante ressaltar que, como o processo é um ciclo, naturalmente ocorre a revisão constante de passos que foram elaborados em fases anteriores. Portanto, é possível adicionar e responder mais Guiding Questions, reescrever a Essential Question, ou até mesmo revisitar a Big Idea,

tudo com o intuito de direcionar melhor a pesquisa e a documentação.

4.1.2 Formulário de pesquisa

Muitas das perguntas da planilha mencionada no tópico 4.1.1 puderam ser respondidas consultando materiais acadêmicos e a internet. No entanto, algumas delas precisariam ser respondidas através de uma pesquisa mais específica, como dúvidas sobre o conhecimento do potencial público-alvo relacionado a aplicações que utilizam *AR*. Portanto, elaborei um formulário com perguntas dinâmicas sobre *AR*, hábitos de jogar em smartphones e carros, além de espaços para deixar sugestões. Para tal, foi utilizada a plataforma *Google Forms*. Com sua divulgação em redes sociais, como no *WhatsApp* (figura 4.2), em junho de 2021, obtive 134 respostas ao longo de uma semana.

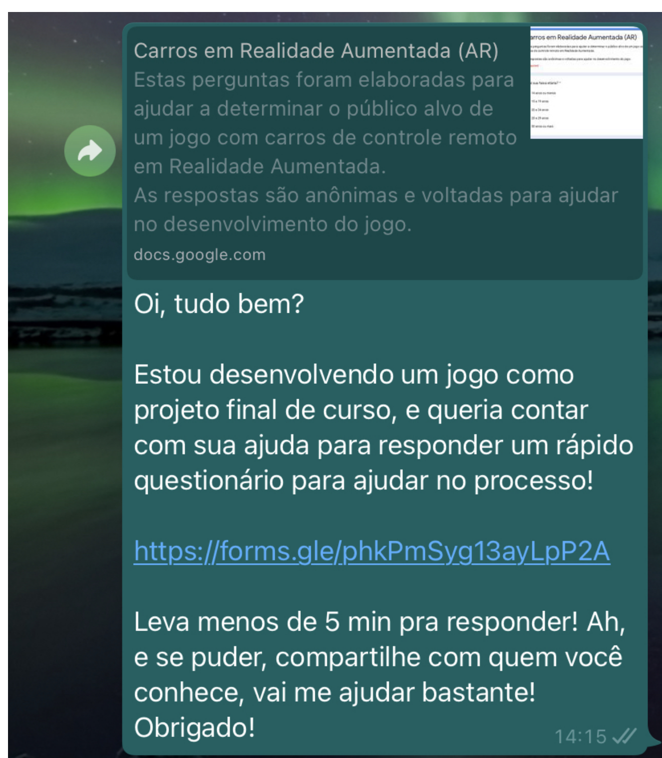


Figura 4.2: Exemplo de mensagem de divulgação da pesquisa no *WhatsApp*.

Ao analisar os resultados da pesquisa, percebi que 116 dos que responderam, o que corresponde a 86,6% do total, já tinha ao menos ouvido falar de Realidade Aumentada. Tal resultado alto era de se esperar, pois muitas das pessoas próximas a meu ciclo social têm contato frequente com novas tecnologias. Porém, aos que não tinham ouvido falar, foi direcionada uma pergunta sobre conhecimento dentre cinco exemplos populares de aplicações que utilizam *AR* (como os mencionados na seção 1.1). Para minha surpresa, 100% destas

pessoas já tiveram contato com pelo menos uma das experiências citadas (era possível deixar a seleção em branco), como mostra o gráfico da figura 4.3.

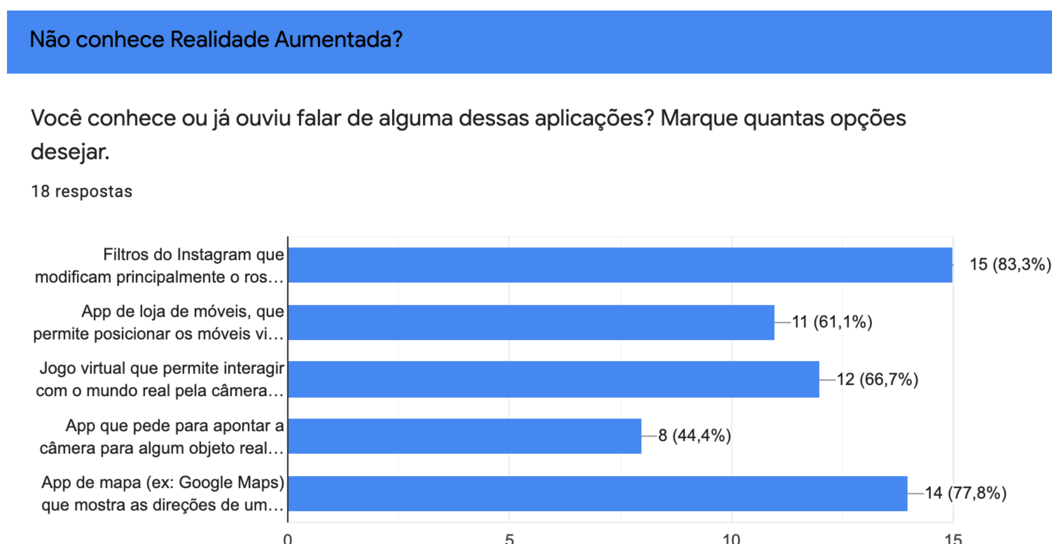


Figura 4.3: Gráfico de respostas sobre conhecimento de aplicações em *AR*.

Além disso, foi possível notar que mais de 80% dos que responderam já haviam tido experiências com carrinhos de controle remoto reais e jogavam jogos eletrônicos no celular, mesmo que com pouca frequência, demonstrando que há um potencial público-alvo para um jogo como o deste trabalho. O formulário poderia ser respondido anonimamente, e havia um campo opcional para sugestões e dúvidas, em que algumas pessoas deixaram sugestões de funcionalidades e interesse em participar do processo de desenvolvimento. Além disso, havia um campo para deixar um e-mail caso quisesse ser notificado sobre novidades ou potenciais versões betas de teste. Do total de respostas, 83 das pessoas forneceram seus e-mails.

4.1.3 Definição da funcionalidade diferencial

Ao examinar as sugestões anônimas de funcionalidades, algumas me chamaram a atenção. A principal delas foi a seguinte: "Um conceito de [corrida] drag pode ser legal, principalmente com **co-op**. Drift seria legal também e poderia ser usado em espaço menores.". *"Co-op"* é como são referidos jogos cooperativos, em que há mais de um jogador colaborando para alcançar um objetivo comum entre eles.

Foi nesse momento que me recordei de uma conversa que tive em 2019 com Shaan Pruden, Diretora Sênior na *Apple*. Quando a mostrei a ideia de um jogo de carros de controle remoto em *AR*, a principal reação dela foi de querer

jogar com as outras pessoas presentes na sala. Com isso, me veio a ideia de tornar isso uma realidade através de um jogo multijogador.

Portanto, a Essential Question foi atualizada para o seguinte texto: "Como promover uma experiência divertida **compartilhada** de carros de controle remoto em AR?". Da mesma forma, o Challenge foi atualizado para: "Melhorar o projeto Synth Car para se tornar um jogo **compartilhado** de carro de controle remoto em AR"

4.2 Estudos da tecnologia disponível

Em posse do desafio a atacar, procurei sobre a viabilidade de implementação da solução utilizando as bibliotecas nativas do *iOS*.

O *SceneKit* possui uma *API* para simplificar a implementação da simulação física de veículos através das classes `SCNPhysicsVehicle` e `SCNPhysicsVehicleWheel`. Portanto, seria possível utilizar a biblioteca para renderizar a cena e simular as interações físicas entre objetos.

Além disso, analisei alguns exemplos de projetos que utilizam frameworks para conectar e realizar comunicações entre dois ou mais dispositivos. Para isso, foram analisados dois projetos disponibilizados pela *Apple* em seu site para desenvolvedores, apresentados nas subseções 4.2.1 e 4.2.2:

4.2.1 AR Multiuser

O projeto *AR Multiuser*[1] foi elaborado para demonstrar o básico sobre a criação e compartilhamento de cenas e objetos em *AR* entre dispositivos. Ele mostra como obter as informações de mapeamento da cena em um dispositivo e compartilhá-la para os demais utilizando *APIs* da biblioteca *MultipeerConnectivity* (2.1.6), fazendo que todos os dispositivos consigam reconstruir localmente a cena criada a partir de um mesmo ponto de origem (x, y, z). A obtenção da cena é feita a partir de uma *API* do *ARKit* que está disponível para uso a partir do *iOS* 12.0.

Vale ressaltar que as informações sobre o ambiente no qual a cena 3D está posicionada incluem imagens de tal ambiente para ser sincronizado com os outros celulares. Ou seja, essas informações podem conter informações sensíveis, portanto devem ser transmitidas entre os dispositivos somente após terem sido criptografadas.

[1] `https://developer.apple.com/documentation/arkit/creating_a_multiuser_ar_experience`

4.2.2 SwiftShot

O jogo *SwiftShot*[2] foi criado para exemplificar a criação de jogos compartilhados em *AR*, similar ao *AR Multiuser*. A principal diferença é que *SwiftShot* apresenta uma maneira de transmitir os dados de física dos objetos e ações customizadas sobre o jogo, como situações de vitória ou interações de um jogador com um objeto da cena. Além disso, ele propõe a utilização de uma arquitetura de projeto e organização de código mais robustas, feitas para promoverem escalabilidade aos jogos digitais.

As figuras 4.4 e 4.5 mostram dois jogadores visualizando a mesma cena e competindo entre si.

Figura 4.4: *SwiftShot* no iPhone 7.

Figura 4.5: *SwiftShot* no iPhone 11.

No documento de explicação sobre o jogo, é mencionado que a experiência foi projetada para encorajar o movimento do jogador pelo espaço físico com o intuito de ser imersiva e engajadora. Porém, deve ter curta duração, pois, mesmo utilizando movimentos suaves, pode cansar o jogador, além de corroborar com as limitações de hardware, vistas na subseção 2.3.1. Ao testar o jogo por 10 minutos em um *iPhone 7*, que teve anos de uso (figura 4.4), a bateria do celular caiu de 85% para 35% de carga, ou seja, utilizou metade

[2]`https://developer.apple.com/documentation/arkit/swiftshot_creating_a_game_for_augmented_reality`

da bateria dele. O mesmo teste foi realizado em um *iPhone 11* (figura 4.5, o qual usou 5% de sua carga no mesmo tempo (de 85% para 80%). Ambos os dispositivos utilizavam a versão 14.4 do *iOS*.

4.3 Fluxos de trabalho

Alguns processos de desenvolvimento, que utilizam os softwares referidos na seção 2.2 foram documentados para referência futura.

O processo de desenvolvimento de modelos 3D do jogo, como o carro e as cenas, começa no *MagicaVoxel*. Terminado o design do modelo, ele e suas texturas são exportados no formato de arquivo com extensão *.obj* e (para o modelo) e *.png* para as texturas. Depois, o arquivo *.obj* é importado pelo *Blender* para ter sua escala ajustada em relação às medidas do mundo do jogo. Então, o modelo é novamente exportado no formato COLLADA[3], com extensão *.dae*. Este arquivo é, então, importado pelo *Xcode* e posto dentro de uma pasta com extensão *.scnassets*, que é uma pasta especial utilizada pelo *Xcode* para guardar assets utilizados pelo *SceneKit*. No entanto, o *SceneKit* trabalha melhor com arquivos de cena com extensão *.scn*. Para isso, o *Xcode* possui uma ferramenta de linha de comando que converte arquivos *.dae* para *.scn*. Então, por fim, é utilizado esse recurso para fazer a transformação final para o tipo *.scn*, o qual poderá ser importado por código para ser exibido na cena em *AR*.

Para o desenvolvimento dos assets 2D da interface, foi utilizado o *Sketch*. Ao terminar o design, é possível exportar os arquivos em formato *.pdf*, que foi escolhido por ser vetorial, o que permite a escalabilidade em inúmeros tamanhos de tela sem perder a qualidade. Arquivos *.pdf* podem ser importados diretamente para o *Xcode*, desde que postos dentro de outra pasta especial, com extensão *.xcassets*.

A música do jogo também teve um fluxo relativamente direto. As composições das duas faixas presentes na versão beta foram feitas utilizando o *GarageBand* e exportadas como arquivos em formato *.mp3*. Esta extensão é diretamente suportada pelo projeto *iOS* no *Xcode*.

Por fim, o fluxo geral de desenvolvimento e aprimoramento de funcionalidades do jogo, ou seja, as alterações no código-fonte em *Swift*, foi feito diretamente utilizando o editor de texto do *Xcode*. Cada funcionalidade veio decorrente de descobertas da fase de investigação do CBL ou de feedbacks de usuários após os testes realizados.

---

[3] `https://www.khronos.org/collada/`

5
Projeto e especificação do sistema

Como mencionado no capítulo 2, o jogo foi inteiramente codificado utilizando a linguagem *Swift*, inicialmente utilizando a versão 4.2 da linguagem e atualizado durante o processo para a versão 5.5.2, que é a mais recente na data de conclusão deste relatório.

O projeto do aplicativo que contém o jogo foi criado utilizando a versão 11.3 do *Xcode*. Este foi atualizado até a versão 13.2, a qual fornece suporte a versão 15.2 do *iOS* 15.2, também as mais recentes na data de conclusão deste relatório.

Considerando as limitações de hardware e software descritas na seção 2.3, foi escolhido que a versão mínima do *iOS* a ter suporte do aplicativo seria a 13.0. Todos os *iPhones* lançados com processador A9 ou superior dão suporte a esta versão. Além disso, de acordo com os dados obtidos até junho de 2021 do site Statista, a fatia de mercado mundial do uso do *iOS* 13.0 ou superior é de 93% da base (Statista 2021), como pode ser visto na figura 5.1. Considerando que no período avaliado não havia sido lançado o *iOS* 15, é possível que este percentual seja ainda maior ao concluir este projeto.

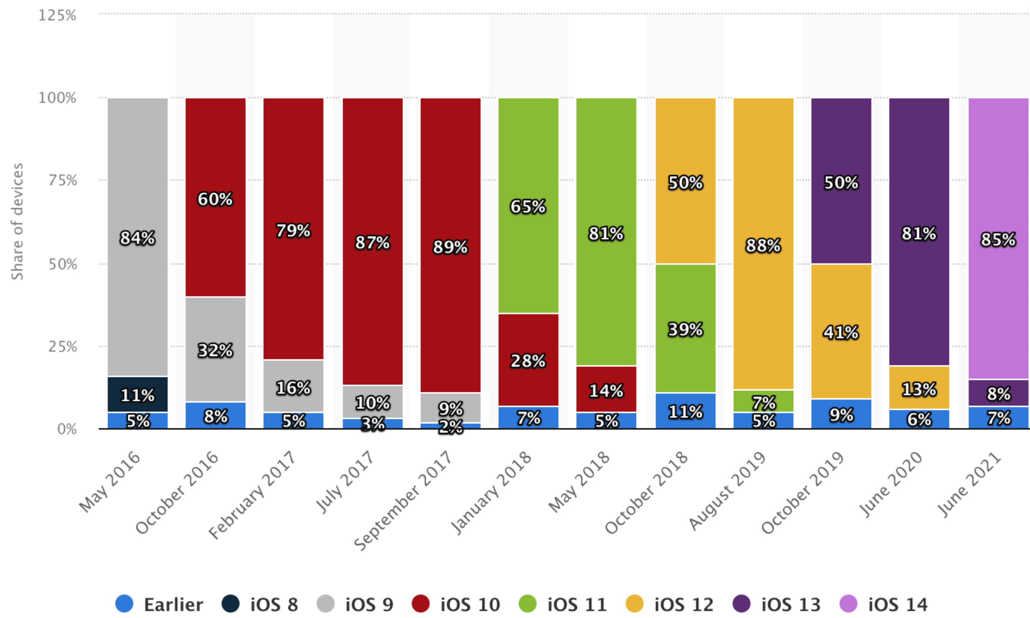


Figura 5.1: Gráfico de quota de mercado das versões do *iOS* (Statista 2021).

5.1 Arquitetura e design de projeto

A organização das classes do projeto tem como objetivo respeitar os cinco princípios do SOLID, os quais foram apresentados originalmente por Robert C. Martin em seu artigo *"Design Principles and Design Patterns"* (Martin 2000).

Para isso, foram escolhidos dois padrões de projeto principais, complementares entre si, além de serem adicionadas camadas conforme a necessidade do projeto.

5.1.1 Model-View-Controller (MVC)

Primeiramente, o jogo é um aplicativo, logo, precisa ter telas e navegação entre elas seguindo as regras de negócio. Para cuidar destas responsabilidades, o aplicativo utiliza o Model-View-Controller (MVC), que é o padrão utilizado pelos projetos de exemplo da *Apple*.

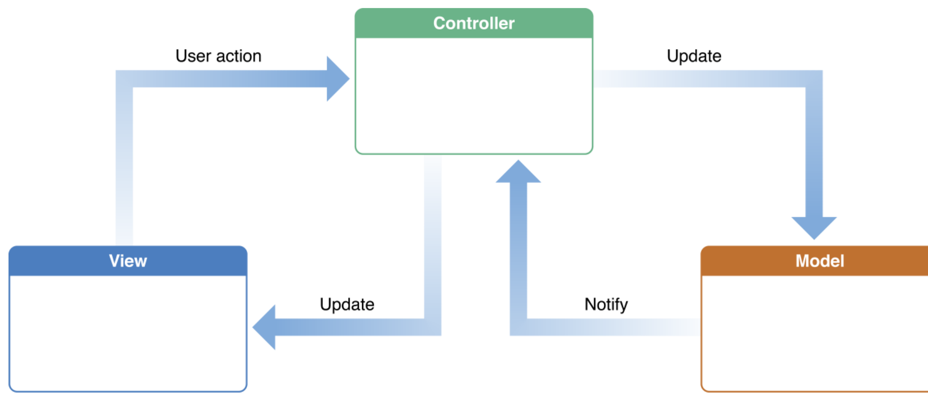


Figura 5.2: Representação visual das camadas do MVC[1].

A camada Model é responsável pelos dados de cada tela; a camada View é responsável pela organização do layout e recepção de eventos do usuário; a camada Controller é responsável por controlar o ciclo de vida da tela, ou seja, os eventos de criação, ativação, inativação, destruição da tela, entre outros. Além disso a Controller cuida da navegação entre telas e serve como intermediária para comunicação entre as demais camadas através das regras de negócio. Ela foi escolhida pela simplicidade e pelo escopo inicial do projeto, que é pequeno se comparado a um aplicativo com diversas telas.

Seria possível separar melhor as responsabilidades da Controller, como, por exemplo, criando uma nova camada para cuidar única e exclusivamente da navegação entre telas, chamada de Coordinator, geralmente utilizado em arquiteturas Model-View-ViewModel (MVVM-C). Mas, como dito, o escopo do projeto prezou pela simplicidade.

A camada de View instancia diretamente objetos do tipo `UIView`, que é a classe disponibilizada pelo *UIKit* para representar o conteúdo visual dentro de uma parte determinada da tela. Também podem ser criadas subclasses a partir de `UIView` para personalizar o comportamento, caso necessário.

Para a implementação da camada Controller, o *UIKit* disponibiliza a classe *UIViewController*, que possui métodos para controlar o ciclo de vida de uma *UIView* associada.

5.1.2 Entity-Component System (ECS)

Além do MVC, outra arquitetura foi escolhida para tratar dos objetos do jogo e das regras geradas pelas interações entre eles. A chamada Entity-Component System (ECS) usa o princípio de composição para as classes do jogo, ao invés de herança. Os modelos que representam objetos do jogo são

[1]Figura: `https://medium.com/ios-os-x-development/modern-mvc-39042a9097ca`

chamados de entidades, e cada entidade é apenas um contêiner que possui uma coleção de componentes. Cada componente representa um dado relacionado às regras do jogo, da simulação temporal/física dele ou outra informação. Por fim, existem sistemas para controlar o fluxo de informações do jogo e aplicarem as regras. Por exemplo: um sistema de pontuação pode atualizar de forma sincronizada a pontuação de cada um dos jogadores a cada frame, ou a cada disparo de eventos do jogo.

Essa abordagem é interessante para ser usada nos componentes do jogo, pois múltiplas entidades podem possuir instâncias diferentes dos mesmos componentes, unificando a lógica de armazenamento daquele dado que o componente representa. Além disso, é possível retirar ou adicionar componentes em entidades dinamicamente em tempo de execução. Um exemplo de uso para essa vantagem da ECS seria caso exista no jogo um sistema de efeitos temporários sobre a entidade controlada pelo jogador, como aumento de ganho da pontuação ou aplicação de vantagens e desvantagens no jogo.

Cada cena do jogo possui um conjunto de entidades e sistemas, que, por sua vez, são coordenadas por uma classe denominada `GameManager`.

A *Apple* providencia um framework voltado para ajudar a criar jogos que utilizam tal arquitetura: o *GameplayKit*, visto na seção 2.1.7 e exemplificado na figura 5.3[2]. Além disso, o projeto *SwiftShot* utiliza este framework para implementar a ECS no desenvolvimento do jogo, e o uso desta arquitetura é muito incentivado na documentação de outros motores gráficos populares para jogos, como o *Unity*[3].

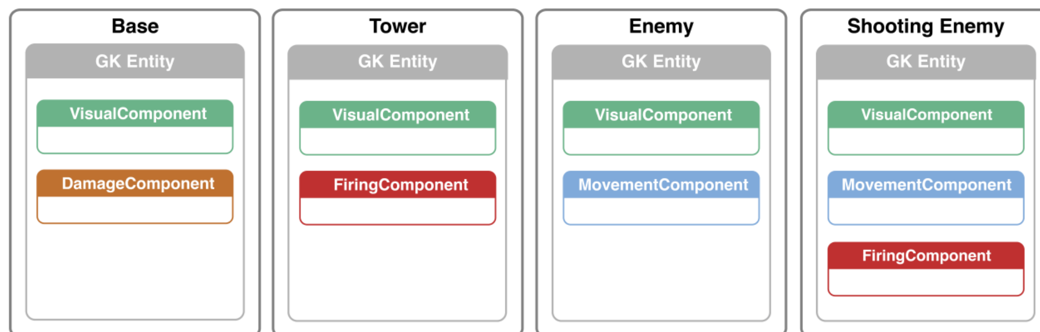


Figura 5.3: Exemplo de arquitetura de um jogo usando ECS e *GameplayKit*.

[2]Imagem: `https://developer.apple.com/library/archive/documentation/General/Conceptual/GameplayKit_Guide/EntityComponent.html`

[3]`https://unity.com/pt`

5.1.3 Networking

Além das camadas dos padrões explicados anteriormente, foi adicionada uma camada responsável pela comunicação entre os dispositivos, chamada de Networking, que se comunica com a camada Controller e com o `GameManager`. Ela tem como dependência as *APIs* do *MultipeerConnectivity*, adaptando as ações de conexão, desconexão e comunicações entre os dispositivos.

Todos os dados que transitam por essa camada são encriptados utilizando a criptografia padrão fornecida pela própria biblioteca *MultipeerConnectivity*, para elevar a segurança do dado em movimento, já que podem conter informações sensíveis, como descrito na seção 4.2.1.

A vantagem de se utilizar uma camada para tal finalidade é que, além de respeitar o princípio da responsabilidade única, facilita uma possível troca de dependência interna. Um exemplo seria se, no futuro, fosse preciso trocar a comunicação entre os dispositivos para ser feita através da internet, ao invés de redes locais, como descrito em 2.1.6. Estando a camada de Networking separada, a interface não precisaria necessariamente ser alterada, pois as mesmas ações de conexão, desconexão e troca de informações já estariam contempladas na interface atual dela.

5.2 Inversão de dependência em Swift

A inversão de dependência é um dos princípios do SOLID, correspondente à letra *"D"*. Em resumo, o princípio diz que o desenvolvimento deve depender de abstrações de alto nível, não de implementações concretas dos módulos. Além disso, as abstrações não devem depender de detalhes, e sim os detalhes devem depender de abstrações (Martin 2000).

A linguagem *Swift* possui um artifício que permite implementar tal princípio: `protocol`, similar a `interface`, da linguagem *Java*.

Como exemplo, o trecho de código a seguir define alguns métodos da abstração da classe `GameBrowser`, que é parte da camada de Network que possui a responsabilidade de encontrar salas de jogos disponíveis e entrar na selecionada. Não é preciso saber da implementação detalhada da classe `GameBrowser` concreta para tê-la como dependência.

```swift
protocol GameBrowserProtocol {
    // ...
    func startBrowsing()
    func stopBrowsing()
```

```
    func join(game: NetworkGame) -> NetworkSessionProtocol?
    func refresh()
}
```

A classe que depende do `GameBrowserProtocol`, parte da camada de Controller, é o `GameBrowserViewController`, que, quando apresentado, mostra e gerencia eventos de uma view de seleção de jogos disponíveis. Para adicionar o `GameBrowserProtocol` como dependência, uma das maneiras possíveis de implementar é injetando-a no construtor do objeto, o que é chamado de **injeção de dependência**.

O código a seguir exemplifica a injeção da dependência na classe da camada Controller:

```
class GameBrowserViewController: UIViewController {

    private let gameBrowser: GameBrowserProtocol

    init(gameBrowser: GameBrowserProtocol) {
        self.gameBrowser = gameBrowser
    }

    // ...
}
```

Com isso, a classe `GameBrowserViewController` necessita de uma instância concreta de `GameBrowser` para ser instanciada:

```
func makeGameBrowserController() -> GameBrowserViewController {
    let gameBrowser = GameBrowser()
    return GameBrowserViewController(gameBrowser: gameBrowser)
}
```

A injeção de dependência facilita, também, a execução de testes unitários automatizados. É possível, por exemplo, criar outra classe concreta que conforme com o protocolo a fim de testar as entradas e saídas dos métodos da Controller.

```
class GameBrowserSpy: GameBrowserProtocol {
    // ...
    func startBrowsing() { /* ... */ }
    func stopBrowsing() { /* ... */ }
```

```
    func join(game: NetworkGame) -> NetworkSessionProtocol? {
        /* ... */
        return nil
    }
    func refresh() { /* ... */ }
}

func makeSpiedGameBrowserController() -> GameBrowserViewController {
    let gameBrowserSpy = GameBrowserSpy()
    return GameBrowserViewController(gameBrowser: gameBrowserSpy)
}
```

5.3 Game design

O jogo *Synth Car*, em sua primeira versão para testes, possui 2 níveis. O primeiro é situado em um estacionamento de um restaurante, em que o objetivo é estacionar o carro dentro de uma das vagas disponíveis, sem tempo limite para tal. O segundo nível é situado em uma pista de corrida, e seu objetivo é completar uma volta no circuito, também sem tempo limite.

Ambos os níveis podem ser jogados no modo *"Singleplayer"*, que significa "único jogador". Como o próprio nome diz, o jogador brinca sozinho e só há um único carro controlável na cena.

Ao iniciar a sessão *singleplayer*, o jogador precisa mapear o ambiente para inicializar o sistema de rastreamento do *ARKit*. Tendo mapeado, o jogador recebe um aviso para posicionar a cena e tocar na tela quando ela estiver em um local conveniente para começar o jogo. No momento que a cena é posicionada, os controles do carro aparecem na tela, assim como um aviso do objetivo principal do nível. Ainda não é possível escolher com qual carro o jogador entrará na sessão.

O jogo também conta com uma versão *"Multiplayer"*, que, inicialmente, suporta até 2 jogadores simultaneamente. Somente o nível do estacionamento está disponível por enquanto para testar o modo *multiplayer*, como mostram as figuras 5.4 e 5.5. Mais detalhes sobre o funcionamento do *multiplayer* serão discutidos na seção 6.1.

Figura 5.4: *Synth Car* no iPhone 7.

Figura 5.5: *Synth Car* no iPhone 11.

Além dos dois modos, há uma terceira cena no jogo, que tem como objetivo mostrar o carro do jogo em mais detalhes. O jogador pode movimentá-lo e alternar a cor da textura do carro entre preto e branco.

6
Implementação e avaliação

O aplicativo *Synth Car* foi inicialmente implementado com base na adaptação para *Swift* de um projeto fornecido pela *Apple* para exemplificar as funcionalidades da *API* de veículos do *SceneKit*[1], pois foi escrito originalmente na linguagem *Objective-C*. Depois, as funcionalidades de *AR* foram sendo implementadas. Posteriormente, o jogo foi sendo incrementado para ter a função multijogador, a partir de adaptações dos projetos mencionados na seção 4.2. As seções seguintes destacam algumas das implementações mais fundamentais do jogo.

6.1
Compartilhamento de mundo

Para compartilhar o mundo, a abordagem tomada foi muito similar à implementada no *SwiftShot*. Primeiro, é preciso que um dos jogadores inicie a sessão no modo host. Este jogador, então, será o responsável por determinar a posição do nível escolhido no ambiente, de maneira similar à implementada no modo *singleplayer*. A partir do momento que o nível estiver posicionado no host, o jogo começa e a camada de Networking recebe uma requisição para começar a anunciar que a sessão está disponível para receber outros jogadores.

Para controlar os estados da sessão no host, foi implementada uma máquina de estados, que é atualizada de acordo com as informações da sessão. A figura 6.1 exemplifica os estados da sessão para o host e as transições entre eles.

[1]Fonte: `https://bitbucket.org/girobai/swiftscenekitvehicledemo/src/master/`

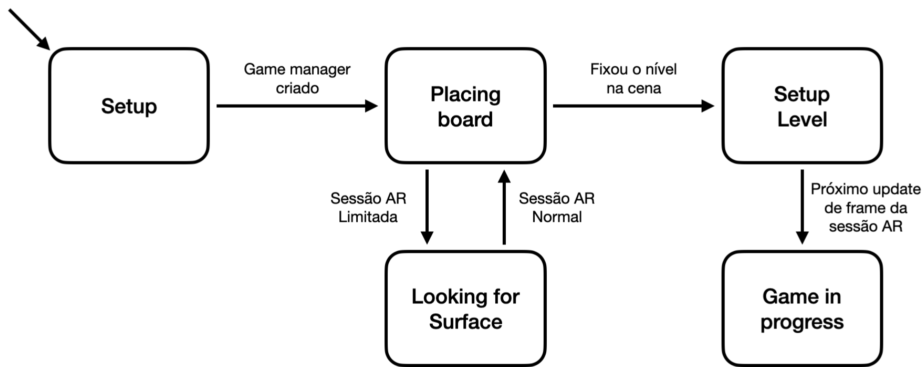


Figura 6.1: Máquina de estados da sessão do host e transições.

Os demais jogadores, chamados de peers, ao entrar na sessão, precisam sincronizar o mundo com o do host. Ao terminar a sincronia, a cena aparece automaticamente e o peer pode começar a controlar seu próprio carro, além de ver os dos outros jogadores sendo atualizados em tempo real. Tal processo também é controlado por uma máquina de estados, que tem alguns dos mesmos estados presentes na do host, como mostrado na figura 6.2.

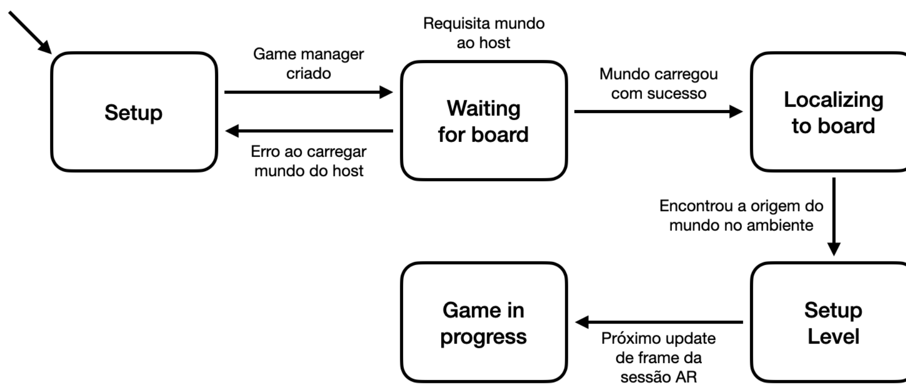


Figura 6.2: Máquina de estados da sessão do peer e transições.

6.2 Compartilhamento da física

Da mesma forma que no compartilhamento de mundo, o jogo *Synth Car* aborda o compartilhamento de física dos objetos da cena de forma semelhante ao *SwiftShot*.

Nem todos os objetos precisam ter suas informações atualizadas entre os dispositivos. No caso deste jogo, apenas o posicionamento do carro do próprio jogador precisa ser compartilhado com os demais dispositivos. Logo, os demais dispositivos conectados precisam enviar o posicionamento de seus respectivos carros.

O posicionamento e direcionamento de cada objeto são definidos por quatro dados em cada instante: posição (vetor tridimensional), orientação (quaternion), velocidade (vetor tridimensional) e velocidade angular (vetor de quatro dimensões). Além disso, é preciso enviar um dado que identifique a qual objeto pertencem as informações já descritas. No caso dos dados do carro, cada carro possui uma propriedade que identifica o jogador a quem pertence, logo é esse o identificador enviado com os dados mencionados.

A cada instante, estas informações precisam ser sincronizadas. Por isso, o envio de dados atualizados da física do próprio jogador ocorrem a cada atualização de quadro do motor gráfico. Logo, enquanto o dispositivo está com a temperatura sob controle, a cena roda a 60 FPS, o que produz 60 atualizações por segundo do posicionamento dos carros. A partir do momento que o celular esquenta demais, a taxa de quadros por segundo reduz para 30 FPS, diminuindo também a frequência de atualização da física entre os dispositivos.

Tais dados são enviados com bastante frequência e possuem mais precisão do que o necessário para os cálculos. Portanto, há oportunidades de otimização futura da codificação e decodificação desses dados para diminuir o volume de dados transmitidas por segundo, ou então aumentar o número de informações passadas por segundo mantendo a mesma taxa de transmissão de bytes por segundo.

No momento que o dado de um objeto chega, ele não será aplicado diretamente. Ele é primeiramente armazenado em uma lista, à medida que forem chegando novos dados. As informações de cada carro são aplicadas somente no momento que o motor gráfico atualiza o quadro da cena.

6.3 Problemas com a simulação veicular

Como mencionado na seção 4.2, a simulação física do *SceneKit* disponibiliza uma *API* para facilitar o desenvolvimento de veículos na cena. No entanto, essa *API* apresenta um problema crítico: não é possível escalar o tamanho do carro na cena, pois a simulação ignora a mudança na propriedade de escala dos modelos 3D.

Ao invés de reduzir ou aumentar ou aumentar os corpos físicos atrelados ao carro de acordo com a escala do modelo, a física é simulada como se a escala estivesse mantida como um vetor unitário (x = 1, y = 1, z = 1). Isso faz com que o resto da cena seja escalado corretamente, porém o veículo não. Então, por exemplo, quando a cena é reduzida, o veículo fica gigante.

Além disso, modelos de carros com tamanhos muito pequenos (ex: com

aproximadamente 30cm de comprimento, que é típico de carros de controle remoto reais) fazem a simulação quebrar por completo. Com essa escala sendo colocada como unitária, a simulação não consegue manter as rodas do carro presas ao chassis na posição correta, e, em alguns momentos, faz o carro voar, como se não houvesse gravidade. Isso me faz pensar que a *API* foi desenvolvida para simular apenas veículos com proporções próximas às reais, o que não seria um problema se fosse possível escalar a visualização dos objetos mantendo as propriedades físicas com valores de um carro real.

6.4 Testes

Para testar as funcionalidades do jogo, é sempre preciso de dispositivos físicos, pois experiências *AR* requerem a utilização da câmera durante toda a sessão. Isso dificultou os testes, pois não é possível utilizar o simulador de *iPhone* e nem sempre eu tinha em mãos um celular carregado com bateria suficiente para rodar uma sessão de *AR*. Além disso, a partir do momento em que precisei testar as funções compartilhadas, o problema aumentou, pois agora é necessário ter em mãos pelo menos dois *iPhones* que tenham suporte a *ARKit*, com bateria suficiente, sem contar que é preciso instalar o aplicativo em dois celulares a cada mudança no código.

A situação mundial também não ajudou nos testes. Como o jogo foi aprimorado durante o ano de 2021, a pandemia da doença COVID-19 impediu que testes grandes pudessem ser executados. Experiências compartilhadas em *AR* precisam que as pessoas testando estejam no mesmo ambiente, o que nem sempre foi possível. Além disso, a fase de testes de baixa fidelidade, que seria executada para descobrir novas ideias sobre objetivos e regras do jogo, precisou ser cancelada pelo mesmo motivo.

Para aumentar o número de pessoas que poderiam testar versões beta do aplicativo, precisei lançar uma versão na plataforma oficial da *Apple* para tal fim, chamada de *TestFlight*. Ela faz parte do fluxo de subida de aplicativos na *AppStore*, e é recheada de funcionalidades que ajudam a receber feedbacks, como registrar novos testadores a partir de links compartilhados ou cadastro por e-mail, além de cada testador poder enviar diretamente comentários, dúvidas e sugestões pela plataforma, dentre outras.

Para ter acesso ao *TestFlight*, precisei assinar o *Apple Developer Program*, na modalidade individual, que custa U$99,00 por ano, pago em moeda internacional. Esta assinatura também dá direito a publicar o aplicativo na *AppStore*, caso este seja aprovado pelo time de revisão de apps.

6.4.1
Testes em celulares diversos

O jogo foi testado com sucesso em vários dos celulares que dão suporte ao *ARKit*. Desde o *iPhone 6S*, o mais antigo, até o *iPhone 12 Pro*, que é um dos mais recentes. No entanto, os testes de experiências compartilhadas foram realizadas apenas em dispositivos rodando *iOS* 14 e 15. Não foi possível encontrar um dispositivo que suporta *ARKit* e ainda estivesse rodando uma versão do *iOS* 13.

6.4.2
Testes de interface gráfica e funcionalidades do jogo

Os testes feitos através do *TestFlight* geraram alguns feedbacks interessantes. Os seguinte pontos foram levantados:

– Os controles do carro precisam ser revistos. O *joystick* da esquerda representa o esterçamento do carro, enquanto o da direita representa a aceleração, para frente ou para trás. A interface foi inspirada em controles remotos reais, no entanto a maioria das pessoas não parecia entender o funcionamento, mesmo depois receberem explicações;

– O sistema de posicionamento do mundo não estava sendo utilizado como o esperado. Algumas pessoas estavam posicionando a cena do estacionamento alto demais, o que dificultava a noção de profundidade dela;

– A maioria dos jogadores sentiu a falta de um tutorial. Aqueles que me conheciam pessoalmente vieram pedir ajuda de como fazer para jogar;

– A interface dos menus do jogo não recebeu reclamações quanto a navegação entre as telas, porém precisa ser aprimorada para seguir a temática visual do resto do jogo;

– Os níveis estão muito grandes proporcionalmente em relação aos objetos do mundo real. Então, um objeto virtual que parece estar perto, na verdade pode estar bem distante;

– As luzes de farol do carro contribuem mais do que o esperado para a imersão do jogador no mundo do jogo;

– Ao objetivo do nível ser cumprido, não há uma mudança significativa na interface para refletir tal feito;

– Alguns jogadores parecem esquecer que é preciso movimentar o próprio celular para movimentar a câmera do jogo, já que elas são a mesma entidade;

- Algumas vezes, numa partida multijogador, ocorre um bug que faz com que o carro de um dos jogadores desapareça da tela. Um botão de reposicionar o carro ajudaria com isso, até que seja consertado;
- Quando um novo jogador entra numa sessão iniciada, é possível que o novo carro apareça na mesma posição de um que já exista na cena, ocasionando no novo carro sendo posicionado acima do primeiro;
- O jogo não ensina ainda como o jogador deve movimentar o celular antes de conseguir fixar o nível ao ambiente.

6.4.3 Testes unitários automatizados

O aplicativo ainda não tem testes unitários automatizados dos componentes do jogo, mas está preparado para recebê-los. A utilização dos princípios SOLID (Martin 2000), especialmente do mencionado na seção 5.2, facilitarão a posterior implementação.

7
Considerações finais

Originalmente, o projeto deveria contemplar o desenvolvimento até o lançamento do aplicativo na *AppStore*. No entanto, mesmo com o escopo do trabalho tendo sido reduzido algumas vezes durante o processo, o resultado continua sendo interessante e inovador. Na *AppStore*, existem jogos de carros de controle remoto, jogos multijogador e jogos que utilizam *AR*. Nenhum, porém, juntou os três conceitos ainda com sucesso. Isso me anima para dar sequência no projeto, pois é um diferencial que ainda tem potencial de ser explorado.

Os feedbacks coletados continuam a me dar muitas ideias de funcionalidades e maneiras de engajar os jogadores em uma experiência completa. Ainda há muito o que possa ser aprimorado, e, para começar, o melhor curso de ação é começar atacando os problemas apontados nos feedbacks da seção 6.4.2.

Vale ressaltar que durante todo o projeto houve a aplicação de conceitos aprendidos durante a faculdade e durante o programa de aprendizado *Apple Developer Academy | PUC-Rio*, o qual participei durante dois anos. Lá, aprendi muito do que eu sei sobre metodologias de pesquisa, *design thinking*, *Swift* e desenvolvimento de aplicativos no geral, e por isso sou muito grato.

Este projeto pode ser visto não somente como um estudo sobre um aplicativo de jogo mobile, mas também como um desafio pessoal sobre como criar um jogo para *iOS* utilizando somente *APIs* disponíveis em bibliotecas nativas do sistema.

Referências bibliográficas

[Abylight 2020] ABYLIGHT STUDIOS. **RC Club 2.0. is now available on the App Store!**, 2020. [Online; acesso em 10 de abril de 2021].

[Apple 2008] APPLE, INC.. **Apple Classrooms of Tomorrow—Today**, 2008. [Online; acesso em 20 de dezembro de 2021].

[Apple 2021] APPLE. **WWDC 2021**, 2021. [Online; acesso em 10 de abril de 2021].

[Azuma 1997] AZUMA, RONALD T.. **A Survey of Augmented Reality**. Presence: Teleoperators and Virtual Environments, 6(4):355–385, August 1997.

[EA 2019] ELECTRONIC ARTS. **Need For Speed Heat Studio**, 2019. [Online; acesso em 10 de abril de 2021].

[Martin 2000] MARTIN, R. C.. **Design principles and design patterns**. Object Mentor, 1(34):597, 2000.

[Merriam-Webster 2021] MERRIAM-WEBSTER DICTIONARY. **Voxel**, 2021. [Online; acesso em 15 de abril de 2021].

[NaturalMotion 2015] NATURALMOTION GAMES LIMITED. **CSR Racing 2 - AppStore**, 2015. [Online; acesso em 10 de abril de 2021].

[Statista 2021] STATISTA. **Share of Apple iPhones by iOS version worldwide from 2016 to 2021***, 2021. [Online; acesso em 20 de dezembro de 2021].

[TNW 2014] TNW. **Apple announces Swift, a new programming language for iOS and OS X**, 2014. [Online; acesso em 15 de abril de 2021].

[The Challenge Institute 2018] THE CHALLENGE INSTITUTE. **Challenge Based Learning - Framework**, 2018. [Online; acesso em 15 de abril de 2021].

[Zendle, Cairns 2018] ZENDLE, D.; CAIRNS, P.. **Video game loot boxes are linked to problem gambling: Results of a large-scale survey**. PloS one, 13(11):e0206767, 2018.