Vinicius Jorge Vendramini

EngineKit

São Paulo

2015

EngineKit

Vinicius Vendramini

Trabalho de Conclusão de Curso apresentado ao Instituto de Matemática e Estatística da Universidade de São Paulo como requisito parcial à obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Flávio Soares Corrêa da Silva

São Paulo

2015

Agradecimentos

À minha mãe, meu pai e meus avós pela confiança, e ao meu irmão pela companhia.

Ao meu orientador, Flávio, tanto pela supervisão e orientação quanto pela confiança e independência.

Ao professor Alfredo pelos conselhos e pela assistência, e aos professores Nina e Gubi, por tornarem este trabalho possível frente às circunstâncias inusitadas.

Table of Contents

1. Introdução

O uso da computação como ferramenta em produções de caráter gráfico e visual tem sido feito há décadas, por cientistas e artistas que se baseiam em cálculos para criar conteúdos estéticos de diversos tipos. Atualmente, a área conhecida como computação gráfica é objeto de diversas pesquisas, tanto acadêmicas quanto industriais, ao redor do mundo. Sua relevância têm sido intensificada pelo crescente potencial computacional de novos processadores, que permitem obter resultados cada vez mais rapidamente e com maior qualidade, e pela popularidade de indústrias que utilizam suas técnicas, como a do cinema e a do entretenimento eletrônico.

Pesquisas nessa área têm resultado na concepção de diversas ferramentas, criadas para que programadores e artistas tenham acesso aos algoritmos e técnicas desenvolvidos de forma prática, sem que sejam necessários conhecimentos avançados sobre o estado da arte. Como é comum na computação, muitas dessas ferramentas acabam por ser disponibilizadas ao público e até ser desenvolvidas de forma aberta. Estas atualmente se espalham pelo espectro de nivelamento, indo desde opções de baixo nível e mais próximas a instruções da CPU e da GPU (como o OpenGL e o DirectX) até alternativas de alto nível com funcionalidades robustas (como a Unreal Engine e a Unity).

A disseminação pública dessas ferramentas tem resultado em um aumento considerável no número de desenvolvedores que as utilizam. Em paralelo à computação gráfica, a computação móvel vem sofrendo um processo similar – e possivelmente mais intenso – que está correlacionado. Desde sua criação em 2008, o número de desenvolvedores registrados na App Store (loja de aplicativos da plataforma móvel iOS, da Apple) tem crescido em ritmo acelerado, e atualmente ultrapassa os 480 mil (POCKETGAMER, 2015); dentre os mais de dois milhões de aplicativos disponíveis na loja, 450 mil (22,4%) são categorizados como jogos – formando a categoria com mais aplicativos da loja e a que conta com o maior apoio de ferramentas da computação gráfica – com uma média crescente de mais de 10 mil

jogos sendo adicionados ao catálogo por mês nos últimos dois anos (POCKETGAMER, 2015).

Esse crescimento no número de aplicativos e de desenvolvedores traz, consequentemente, uma grande quantidade de desenvolvedores novatos, procurando se aproveitar das oportunidades apresentadas. Isso é evidenciado pela crescente abundância e popularidade de guias presentes na internet para ajudá-los e instruí-los em seu ingresso na plataforma. Soma-se a isso o fato de que, atualmente, 93% dos programadores têm sistemas móveis entre suas plataformas alvo (STATISTA, 2014) (um ambiente que era desprezível antes da abertura do mercado de smartphones, há sete anos) e conclui-se que há uma grande quantidade de programadores inexperientes no desenvolvimento para plataformas móveis, que podem se interessar pelo desenvolvimento de jogos — uma vez que eles constituem a maior categoria entre tais aplicativos.

Nessa conjuntura, seria esperado que houvesse acontecido um esforço por parte dos desenvolvedores de ferramentas gráficas dirigido a esse público. Na realidade, encontra-se aqui um vácuo tecnológico. Comparado com o ambiente de sistemas operacionais em computadores, as plataformas móveis apresentam maior escassez de tais programas — e os que existem apresentam problemas que os tornam inadequados para novatos.

Alguns, como a Unreal Engine e a Unity, são programas complexos e robustos, muitas vezes desenvolvidos para profissionais. De acordo com levantamentos feitos em fóruns de discussão sobre esses programas, alguns problemas os tornam aversivos a novos usuários: o fato de apresentarem diversas funcionalidades, que precisam ser aprendidas antes que se possa usar o programa propriamente; o fato de exigirem conhecimentos sobre programação, em particular de linguagens de baixo nível; o fato de ocasionalmente serem financeiramente inacessíveis; entre outros. Outras ferramentas, como o OpenGL, foram desenvolvidas a fim de promover grande versatilidade e alta performance a seus usuários, sacrificando, para tanto, a sua facilidade de uso.

A dificuldade de acesso aos programas citados também é evidenciada na abundância de recursos voltados para seu aprendizado. Pesquisas simples revelam diversos textos, artigos e vídeos visando ensinar desenvolvedores novos a usar tais tecnologias. Esse conteúdo frequentemente é extenso, o que destaca as complexidades inerentes, e ocasionalmente (como no caso da Unreal Engine) é criado pelos mesmos desenvolvedores da ferramenta, evidência de que os próprios reconhecem as dificuldades envolvidas no seu uso.

Assim, torna-se importante desenvolver um sistema novo como alternativa às ferramentas citadas, que minimize a necessidade de conhecimentos externos. Dessa forma, será possível dar a usuários a oportunidade de focar melhor seus esforços no desenvolvimento de seu conteúdo e na expressão de sua criatividade, além de tornar o ambiente mais convidativo a desenvolvedores novos que, de outra forma, poderiam ter sido afastados pelas dificuldades encontradas no começo do processo de desenvolvimento.

Este projeto pretende desenvolver uma ferramenta que auxilie o desenvolvimento de conteúdo gráfico para plataformas móveis, priorizando a acessibilidade a desenvolvedores leigos mas mantendo-se flexível e completa o suficiente para servir como alternativa prática a programas similares. De forma específica, o sistema deve servir como motor gráfico para conteúdos diversos, minimizar a exposição de ferramentas de baixo nível e abstrair conceitos complexos para se manter acessível, e implementar funcionalidades suficientemente variadas e robustas a ponto de se manter útil e competitivo.

Para tanto, o projeto foi desenvolvido na plataforma iOS, fazendo uso de bibliotecas e recursos do sistema operacional para manter o desenvolvimento confiável e em ritmo acelerado (dado o prazo limitado), e se baseou em pesquisas feitas na literatura recente para garantir o uso de algoritmos e paradigmas atuais e confiáveis.

2. Implementação

A implementação do EngineKit foi estudada e planejada. Inicialmente, foi feita uma pesquisa sobre o estado da arte da computação gráfica, a fim de se obter uma compreensão maior sobre o processo de implementação de um motor gráfico. Entre os autores estudados destaca-se Gregory (2015).

Nesse livro, o autor descreve diversos padrões de *design* usados na implementação de motores gráficos, comentando casos gerais e usando uma implementação concreta da qual fez parte como exemplo. Ele também enfatiza a complexidade envolvida quando um projeto envolve criar todos os subsistemas envolvidos na arquitetura de um motor gráfico. Segundo o autor, motores profissionais – que muitas vezes re-implementam mesmo os algoritmos mais fundamentais para maximizar sua performance – frequentemente fazem uso de uma gama de bibliotecas para facilitar e agilizar o desenvolvimento da ferramenta (GREGORY, 2015).

Gregory se utiliza da figura 1 como ponto de partida para descrever detalhes da implementação de cada sistema apresentado. Em suas palavras:

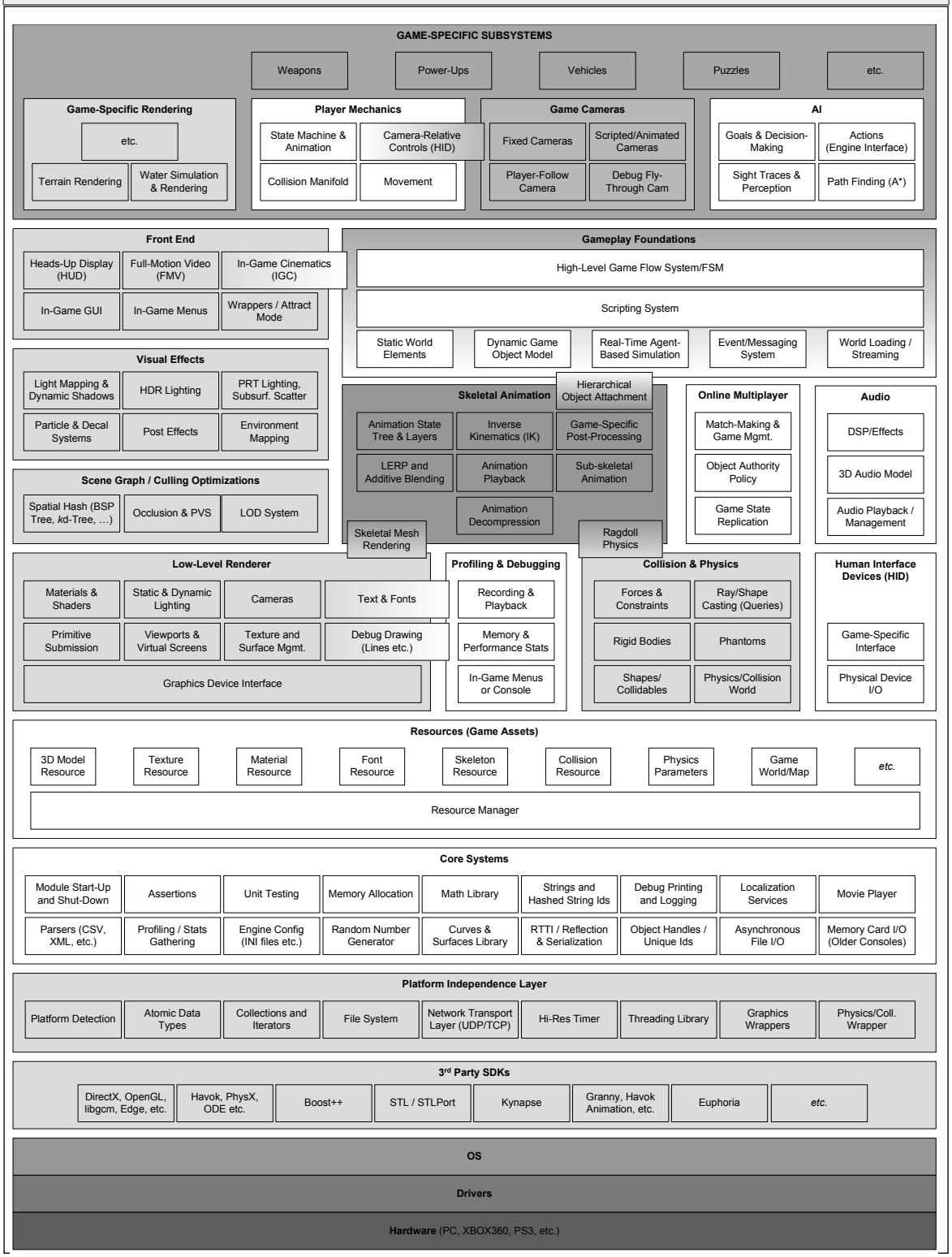
> [A figura] mostra todos os grandes componentes em tempo de execução que compõe um típico motor gráfico 3D. Sim, é *grande*! E este diagrama nem inclui todas as ferramentas. Motores gráficos são, com certeza, grandes sistemas de *software* [Tradução do autor] (GREGORY, 2015, p.32)[1].

Assim, foi necessário decidir quais subsistemas inicialmente seriam essenciais para o desenvolvimento do EngineKit e quais poderiam se apoiar em bibliotecas externas para auxiliar a implementação do projeto. Essas escolhas formaram uma estrutura inicial, que teve de ser adaptada ao longo do desenvolvimento do sistema (resultando na figura 2) mas serviu como norte para diversos desafios e decisões encontrados.

[1] No original: "Figure 1.15 shows all of the major runtime components that make up a typical 3D game engine. Yeah, it's *big!* And this diagram doesn't even account for all the tools. Game engines are definitely large software systems".

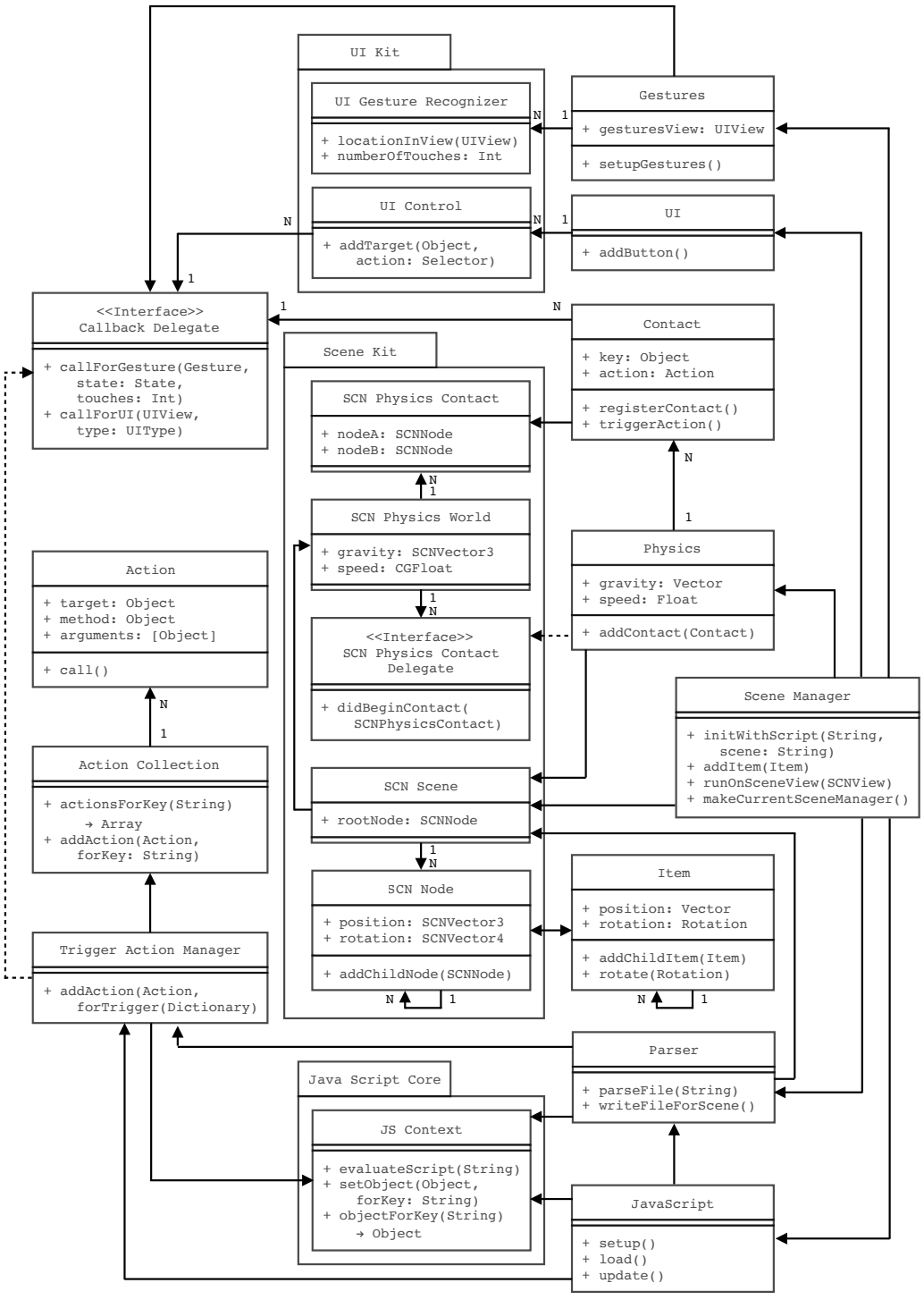
Esta seção descreve as principais características e funcionalidades do sistema, explica algumas das decisões tomadas ao longo de seu desenvolvimento e, de forma geral,

FIGURA 1: Estrutura de um motor gráfico profissional.
FONTE: Gregory, 2015, p.33.



descreve os subsistemas criados e suas conexões.

FIGURA 2: Diagrama na Linguagem de Modelagem Unificada (Unified Modeling Language, ou UML) ilustrando a estrutura geral do EngineKit. Por praticidade, alguns nomes foram simplificados, e foram representadas apenas informações relevantes.
FONTE: Vendramini, 2015.



2.1 Configuração

O EngineKit foi desenvolvido visando ser uma ferramenta quase invisível ao usuário, de modo a permitir que ele se concentre apenas no desenvolvimento de seu conteúdo. Para tanto, sua configuração inicial é curta e tem pouco espaço para erros. Ela envolve duas tarefas separadas: a criação de um aplicativo novo e a adição de componentes do EngineKit.

A criação de um aplicativo para iOS é um processo independente que varia conforme as plataformas envolvidas evoluem. A cada passo, essa evolução geralmente torna o processo mais acessível. Apesar disso, como ele é controlado pelos criadores do sistema operacional, sua acessibilidade é um fator que desenvolvedores independentes são incapazes de influenciar.

A adição de componentes do EngineKit, por outro lado, é um processo que foi desenvolvido com cuidado. O resultado envolve apenas dois passos. Primeiro, deve-se importar a biblioteca, o que pode ser feito de várias maneiras, inclusive com um simples arrasto do arquivo para o projeto. Segundo, deve-se adicionar duas linhas de código, que dizem à biblioteca quais objetos usar como entrada de informações e como saída de conteúdo, respectivamente.

Essa configuração pode ser repetida diversas vezes em aplicativos que queiram criar múltiplas cenas (paralelas ou sequenciais) e pode ser alterada para permitir acesso a configurações diferentes.

Nos aplicativos existentes, o processo completo pode ser um pouco aversivo a amadores — percebe-se, novamente, pela abundância de artigos que procuram explicar como realizar processos similares com outros sistemas. Para contornar esse problema, o repositório do EngineKit inclui um projeto em branco pré-configurado, que pode ser copiado por usuários e permite o começo imediato do seu uso, com configurações padrão.

O desenvolvimento de um aplicativo que usa o EngineKit sem configurações adicionais envolve apenas a criação de dois arquivos, em que o usuário especifica todos os detalhes sobre o conteúdo que quer gerar. Isso inclui informações sobre os

elementos gráficos em si (suas posições, cores, rotações, etc.) e sobre comportamentos relativos a esses elementos (configurações da simulação de física, respostas a gestos do usuário, etc.).

Os formatos e padrões usados nesses arquivos foram pensados de forma a facilitar que usuários os criem e os modifiquem manualmente. Ainda assim, como alternativa a edições manuais, foi criado um aplicativo (desenvolvido para iOS). Esse aplicativo é capaz de ler esses pares de arquivos e exibir o conteúdo resultante, editá-lo e testá-lo, e então sobrescrever o resultado.

O conjunto de interfaces formado pela escrita dos arquivos e a utilização do aplicativo constitui o *front end* da biblioteca, e sua implementação é detalhada no item 2.2.

A parte do sistema responsável por transformar as descrições fornecidas em algo que possa ser mostrado na tela é chamada de *back end*. Ela é composta por alguns subsistemas, cada um responsável por uma funcionalidade chave da biblioteca. O item 2.3 inclui detalhes sobre essa implementação.

2.2 Front End

2.2.1 Estrutura

Dentre os objetivos principais do EngineKit, a meta de ter um sistema acessível a novos usuários foi central para o *design* do *front end*. Mais especificamente, desejava-se apresentar uma interface que evitasse incluir ferramentas de baixo nível.

Para isso, buscou-se inspiração em um paradigma amplamente usado por amadores na programação: o *webdesign*. O desenvolvimento nessa área é comumente caracterizado por duas abordagens: usa-se arquivos em uma linguagem *markup* (como o HTML) para descrever os elementos estáticos de uma página; e arquivos em uma linguagem *script* (como o JavaScript ou o PHP) para fornecer funcionalidades mais dinâmicas e complexas[2].

[2] Nota-se que essa simplificação não é uma descrição exata da área, e busca apenas transmitir os pontos que influenciaram o desenvolvimento do sistema.

Além de ser popular, essa combinação se adapta bem ao desenvolvimento de conteúdo gráfico. É possível usar uma linguagem *markup* para descrever os elementos a serem apresentados em cena, e uma linguagem *script* para descrever seus comportamentos. O paradigma também cuida de esconder quaisquer implementações em baixo nível sob essa camada de abstração, já que o único contato dos usuários com o sistema acontece durante a implementação desses arquivos. A acessibilidade fornecida por essa camada então depende exclusivamente das linguagens escolhidas para cada arquivo.

2.2.2 Linguagem Script

Para a linguagem de programação, escolheu-se o JavaScript. Isso se deu principalmente por dois motivos: quando a escolha foi feita, ela era uma das linguagens mais populares (fator mantido do início ao fim do projeto); e há suporte significativo para seu uso no iOS.

Esse suporte é fornecido pela biblioteca JavaScriptCore. Ela é a biblioteca usada pelo navegador de internet no próprio iOS, o que é evidência da sua qualidade e torna provável que suporte a ela seja mantido por mais tempo (ao menos enquanto o sistema operacional usá-la).

Sua integração com o projeto foi muito simples e foi feita no início do desenvolvimento. Isso permitiu que a sua interface com o resto do sistema fosse criada e testada gradativamente, em paralelo com o desenvolvimento dos outros subsistemas.

A biblioteca toma vantagem de algumas funcionalidades particulares de meta-programação do Objective-C, linguagem em que o resto do sistema foi implementado. Assim, ela garante uma integração profunda entre as duas linguagens, e permite que funcionalidades criadas sejam exportadas com facilidade para o JavaScript. Isso proporcionou mais agilidade durante o processo de implementação e minimiza o esforço para incluir futuras adições e modificações no sistema.

Graças a essa flexibilidade, foi possível disponibilizar ao JavaScript todas as funcionalidades implementadas. Desse modo, está garantido que qualquer tarefa que

precise ser feita com programação possa ser feita interagindo apenas com a camada de alto nível que a linguagem apresenta. Portanto, com possíveis exceções durante o processo inicial de configuração, o usuário pode evitar qualquer contato com ferramentas de baixo nível.

2.2.3 Linguagem Markup

Apesar de esse objetivo ter sido atingido apenas com a linguagem *script*, é necessário notar que ela ainda é uma linguagem de programação. Por mais que ela seja mais acessível que as linguagens apresentadas por sistemas similares, os comentários de usuários dessas alternativas levam a crer que seria melhor minimizar o contato com qualquer tipo de programação, cuja simples presença já envolve algum esforço gasto em seu aprendizado.

Seguindo ainda o paradigma de *webdesign*, vê-se que uma linguagem de *markup* pode contribuir nesse sentido. Essas linguagens são, por natureza, apenas formas de anotação ou descrição de elementos. Por um lado, essa simplicidade pode torná-las mais acessíveis que linguagens de programação; por outro, significa que não incluem diversas funcionalidades presentes em linguagens de programação[3], e portanto não podem ser usadas para substituí-las completamente.

Assim, em vez de procurar uma substituição completa, a linguagem *markup* foi projetada de forma a conter as funcionalidades mais frequentemente utilizadas (e as que melhor se adaptavam ao seu formato), na esperança de se tornar a principal forma de interação com o usuário, tendo a linguagem *script* como complemento para operações que fossem além da sua capacidade.

As linguagens que existiam no começo do projeto se mostraram opções ruins para o uso pretendido. Elas frequentemente eram difíceis de ler (como evidenciado pela abundância de programas usados para manuseá-las[4]) ou não apresentavam a

[3] Linguagens de *markup* dificilmente serão, por exemplo, Turing completas, já que são comumente usadas para descrever dados e não computações. As que fogem à regra o fazem com uma complexidade considerável, o que reforça o fato de serem uma alternativa ruim à linguagens *script*.

[4] O antigo Microsoft FrontPage, o Adobe Dreamweaver, e os mais recentes CoffeeCup e Google Web Designer são alguns exemplos de editores de HTML.

versatilidade necessária para implementar todas as funcionalidades. Assim, decidiu-se pela criação de uma nova linguagem.

As decisões tomadas durante esse processo de criação visaram sempre optar por alternativas que priorizassem a simplicidade, na esperança de assim aumentar a acessibilidade. Algumas dessas decisões incluem o uso de indentação no lugar de chaves (ou outros caracteres) e a implementação de uma gramática particularmente flexível, que será detalhada no item 2.3.9.

A maior parte das funcionalidades exportadas para a linguagem *markup* focam na descrição e composição de objetos na cena, que podem vir a ser arbitrariamente complexos. Elas incluem a criação, composição e adição de objetos, bem como a especificação de suas propriedades físicas e de suas respostas a gestos na tela. Mais detalhes sobre o subsistema responsável pela leitura dessa linguagem estão no item 2.3.8.

2.2.4 Aplicativo Editor de Cenas

Existem outras formas de trazer o desenvolvimento gráfico para um nível mais alto que o da programação, frequentemente presentes nos principais motores gráficos – principalmente naqueles voltados para profissionais. É comum que eles forneçam uma série de ferramentas para manusear diferentes tipos de recursos e arquivos, e entre elas destaca-se frequentemente um editor de cenas.

Esse componente, normalmente disponível para sistemas operacionais de computadores, permite que uma versão estática do conteúdo gráfico seja editada de forma visual. Ele comumente usa gestos do mouse aliados a atalhos do teclado para simular a forma como um ser humano manipularia uma cena, executando rotações, translações, etc com arrastos do cursor.

No EngineKit, essa ferramenta foi implementada como um aplicativo para o próprio iOS. Isso permitiu que ela usasse o próprio sistema para sua implementação, de forma que servisse como teste de aceitação das funcionalidades implementadas até então. Conforme funcionalidades novas eram implementadas no sistema, elas eram também

incorporadas ao aplicativo, garantindo cobertura universal de testes para a biblioteca resultante.

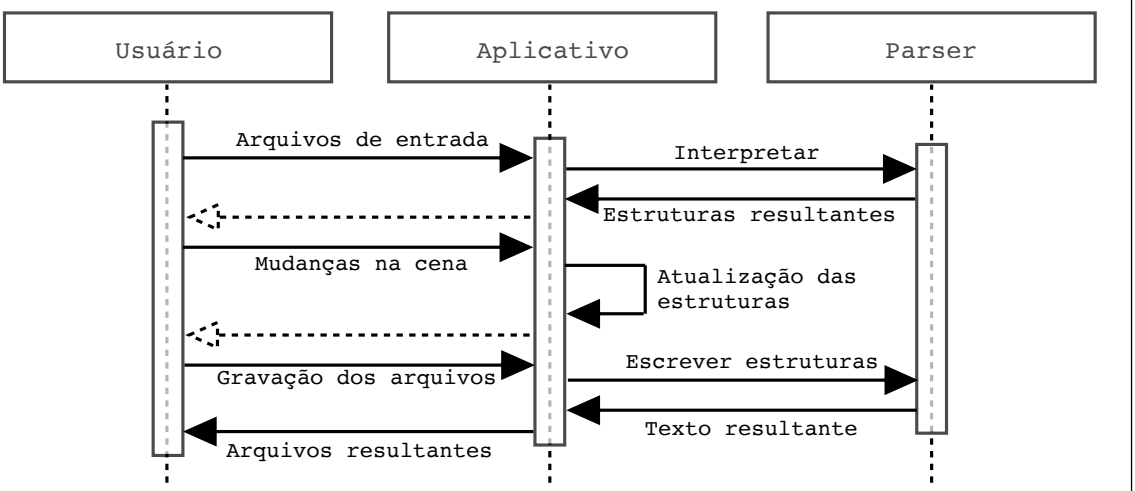
O fato de essa ferramenta ser um aplicativo de iOS (e não um programa para computadores) traz novas possibilidades e limitações ao *design* da sua interface. Por um lado, o fato de as interações serem simulações de gestos humanos (como se alguém segurasse a cena para poder girá-la e movê-la) é mais evidente em uma plataforma móvel, em que gestos são aplicados com as mãos na tela no lugar do uso do mouse. Isso tende a tornar algumas interações mais intuitivas — algo que se espera comprovar e aprimorar com testes de usabilidade (Capítulo 3).

Por outro lado, o uso de atalhos do teclado em programas de computador fornece a eles uma flexibilidade difícil de igualar em um dispositivo móvel. Esse fator foi amenizado pela variedade de gestos suportados pelo sistema operacional: o aplicativo reconhece gestos quando o usuário arrasta, gira, belisca ou apenas clica na tela, e é capaz de contar quantos dedos estão sendo usados, aumentando a diversidade de interações e usando-a para fornecer funcionalidades diferentes.

Outra forma comum de interação com programas de computador envolve o uso de menus, botões e outros elementos de interface para realizar ações mais específicas. Esses elementos puderam facilmente ser implementados usando a interface nativa do iOS. Com eles, o usuário pode selecionar objetos específicos, adicionar e retirar objetos na cena, criar objetos novos, mudar suas propriedades e especificar como cada objeto responde a gestos quando o conteúdo for executado.

Este editor não foi criado como alternativa para o uso dos arquivos citados anteriormente (itens 2.2.1 e 2.2.2), e sim como alternativa para a escrita e edição manual deles. Os arquivos são usados para descrever completamente o resultado final; o aplicativo é capaz de carregá-los, permitir a edição visual de seu conteúdo, a execução de uma simulação do resultado final e que as mudanças feitas sejam sobrescritas nos arquivos originais. Esse fluxo de interações é representado pelo diagrama da figura 3.

FIGURA 3: Diagrama de sequência em UML representando um fluxo de interações entre o usuário e o sistema, com uso do aplicativo.
FONTE: Vendramini, 2015.



2.3 Back End

Para ser capaz de apresentar interfaces mais amigáveis ao usuário, o sistema conta com um back end consideravelmente mais complexo e extenso. Ele está separado em subsistemas, cada um responsável por implementar uma funcionalidade; esses subsistemas então compõem um objeto gerenciador, que é apresentado ao usuário.

Esta seção apresenta cada subsistema e o gerenciador, e fornece alguns detalhes sobre suas implementações, as decisões de *design* envolvidas e as bibliotecas usadas.

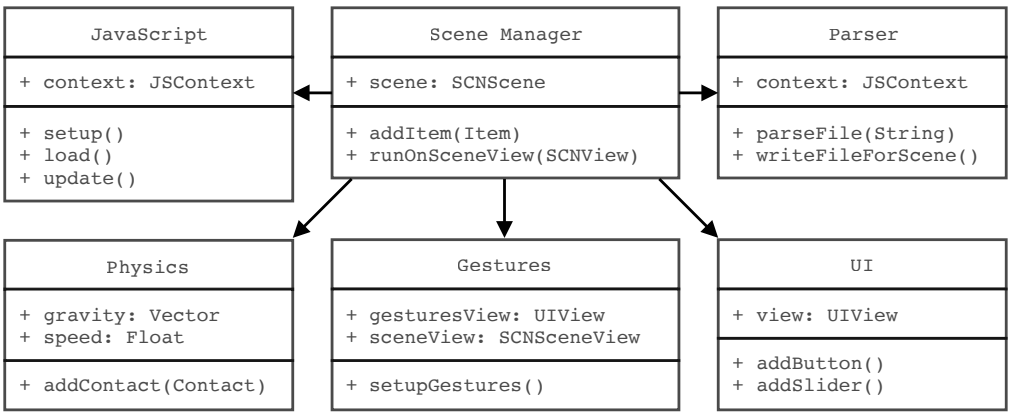
2.3.1 Gerenciador de Cenas

O nome dado ao objeto que gerencia os subsistemas é Scene Manager (gerenciador de cenas)[5]. Ele é responsável por criar os objetos que representam cada subsistema, configurar esses objetos e suas conexões (figura 4).

Esse gerenciador é o objeto que representa a principal forma de interação entre o usuário e o sistema em uma linguagem nativa da plataforma. A configuração inicial da biblioteca, em que são especificados os recursos de entrada e saída, é feita pela criação e configuração do próprio gerenciador.

[5] O código do EngineKit foi escrito com nomes de classe, variáveis e afins e inglês, a fim de manter a mesma língua em que foram escritas as bibliotecas utilizadas e o sistema operacional.

FIGURA 4: Diagrama de classes em UML representando o gerenciador de cenas e os subsistemas que ele é responsável por administrar.
FONTE: Vendramini, 2015.



Para além dessa configuração, o objeto em questão oferece funcionalidades para a manipulação da cena, incluindo métodos para parar e retomar a sua execução e um ponto de partida para a adição de objetos gráficos à raiz da hierarquia (item 2.3.2).

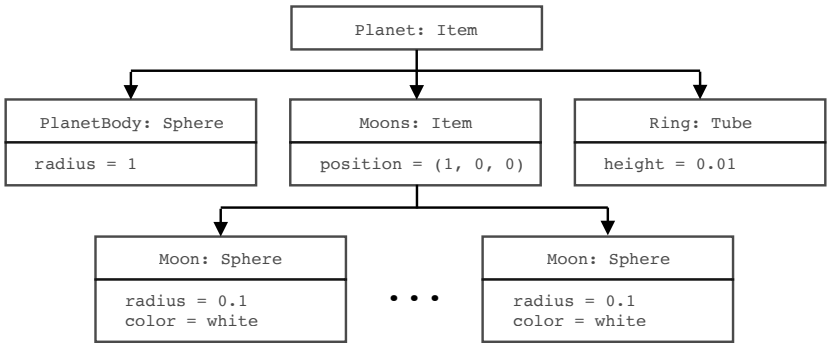
Por fim, como o gerenciador de cenas encapsula objetos que gerenciam todos os outros subsistemas, as suas instâncias são completamente independentes (a não ser por alguns elementos globais que permitem a comunicação entre instâncias). Assim, é possível criar um número arbitrário de gerenciadores para manipular o mesmo número de cenas diferentes simultaneamente, que por praticidade compartilham uma mesma base de conhecimento.

2.3.2 Itens e Modelos

O conteúdo gráfico que é gerenciado pelo EngineKit é composto de objetos geométricos, que são exibidos na cena, e dos comportamentos que são atribuídos a eles. A representação interna desses objetos é responsabilidade de um motor gráfico chamado SceneKit, que o sistema empacota em uma camada de abstração. Esse empacotamento constitui um subsistema que é baseado em paradigmas presentes no SceneKit, de forma a manter sua estrutura.

Em linhas gerais, a biblioteca trata objetos da cena como parte de uma hierarquia conhecida como grafo de cena. Nesse grafo, vértices mais próximos da raiz da cena representam objetos complexos, enquanto as suas folhas representam as formas

FIGURA 5: Diagrama de objetos em UML, representando a hierarquia correspondente a um objeto exemplo.
FONTE: Vendramini, 2015.



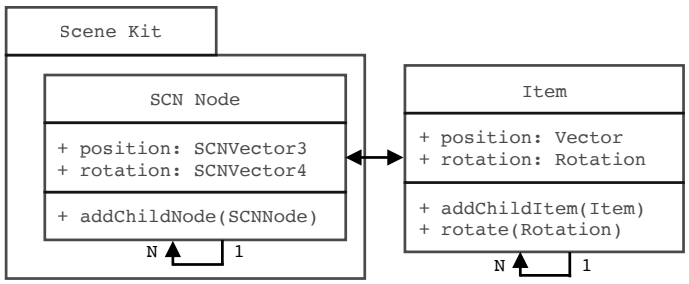
geométricas elementares que constituem esses objetos. A figura 5 mostra o exemplo de hierarquia correspondente a um objeto que se aproveita desse sistema.

Esta implementação em hierarquia apresenta algumas vantagens na criação de cenas complexas. Com ela, é possível especificar propriedades de formas geométricas simples com relação ao objeto que elas devem formar, e então especificar propriedades que afetam o objeto completo com relação à cena que se quer criar. Com isso, existe um controle eficaz (e eficiente) sobre a capilaridade das mudanças que se quer fazer em uma cena.

O empacotamento criado pelo EngineKit sobre essa hierarquia envolve a adição de métodos a várias classes do motor gráfico (para melhor integrá-las ao sistema) e, principalmente, a criação de uma classe nova para empacotar os objetos em cena, chamada de Item.

A classe Item é criada de forma que cada Item corresponda a um SCNNode, nome original dado pelo motor gráfico aos objetos geométricos. Cada item contém uma referência ao seu SCNNode, ao qual ele transmite as operações que forem necessárias (figura 6). Dessa forma, toda a funcionalidade original do motor gráfico é mantida. Além disso, a classe Item extende a funcionalidade de um SCNNode com métodos e atributos próprios. Isso permite que novas ações sejam realizadas na mesma estrutura e que comportamentos originais sejam sobrepostos, tornando o objeto final melhor adaptado ao resto do sistema.

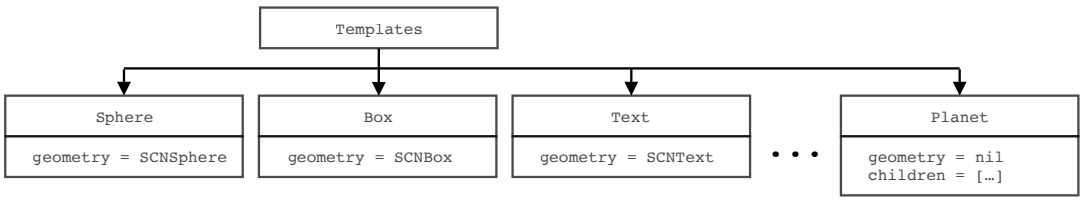
FIGURA 6: Diagrama de classes em UML, representando o empacotamento implementado pela classe Item.
FONTE: Vendramini, 2015.



As estruturas relativas a SCNNodes foram mantidas dentro do motor gráfico: objetos que precisem acessar itens específicos devem percorrer a hierarquia interna de SCNNodes, e então acessar o item respectivo através de uma referência que cada SCNNode tem a seu item. Como a classe Item é externa à biblioteca, essa referência tem de ser inserida por recursos de meta-programação do Objective-C em tempo de execução. Este método cria um ciclo de referências que teve de ser cuidadosamente administrado para evitar vazamentos de memória, mas evita que se tenha de espelhar estas hierarquias externamente — uma solução que seria passível de falhar caso algoritmos internos do motor gráfico fossem atualizados. O acesso às estruturas internas da biblioteca é responsabilidade da classe Item, que então expõe essas funcionalidades de forma segura a outros subsistemas.

Entre as funcionalidades que essa abordagem permite adicionar está a presença de *templates* (ou modelos). Eles são instâncias da classe Item que são mantidas fora da cena, em um registro geral de modelos do programa (figura 7). Uma implementação de cópia profunda de Itens permite que o programa crie novos Itens a partir desses modelos. Assim, o usuário é capaz de construir um modelo complexo e criar múltiplas instâncias a partir dele. Isso evita a duplicação de código e torna a hierarquia resultante mais compreensível, já que ela deixa de ser um conjunto de formas geométricas e passa a ser um conjunto de objetos coesos com algum valor semântico (figura 5).

FIGURA 7: Diagrama de objetos em UML, representando registro global de modelos.
FONTE: Vendramini, 2015.



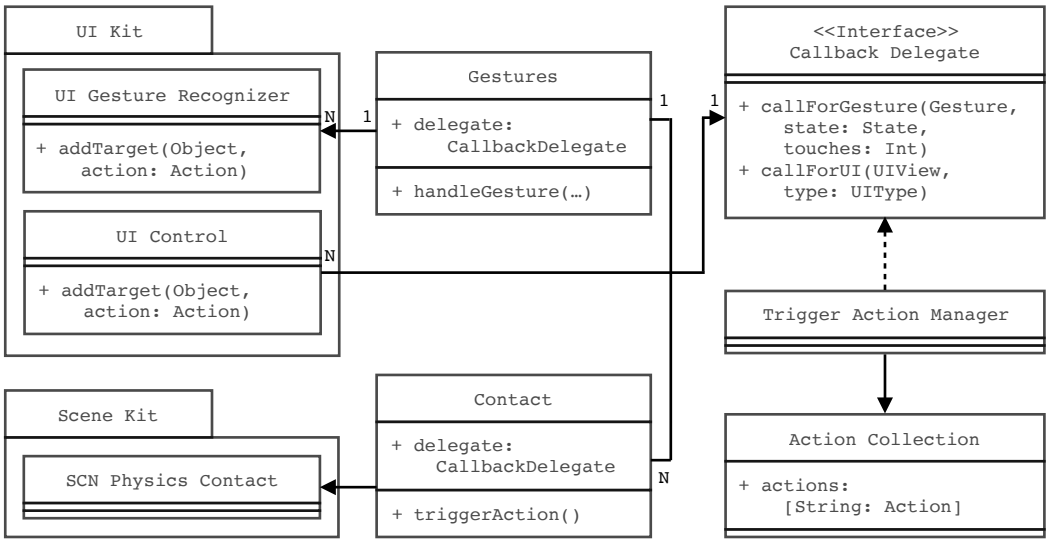
2.3.3 Eventos e Ações

Os comportamentos que podem ser associados a Itens em uma cena são administrados por um subsistema especializado implementado pela classe Trigger Action Manager, ou gerenciador de eventos e ações. Essa classe é responsável por administrar a detecção de eventos e, em cada caso, ativar as ações corretas.

Os eventos são detectados por um conjunto de bibliotecas externas ao EngineKit. Cada uma é responsável por avisar o gerenciador quando um tipo de evento acontece, seja ele um gesto na tela, um acontecimento na simulação de física ou uma interação com um elemento de interface do iOS (como um clique em um botão).

As interfaces apresentadas por essas bibliotecas são similares e são comuns em todo o sistema operacional; elas essencialmente requerem que o gerenciador se cadastre

FIGURA 8: Diagrama de classes em UML, representando as interações do gerenciador de eventos e ações.
FONTE: Vendramini, 2015.



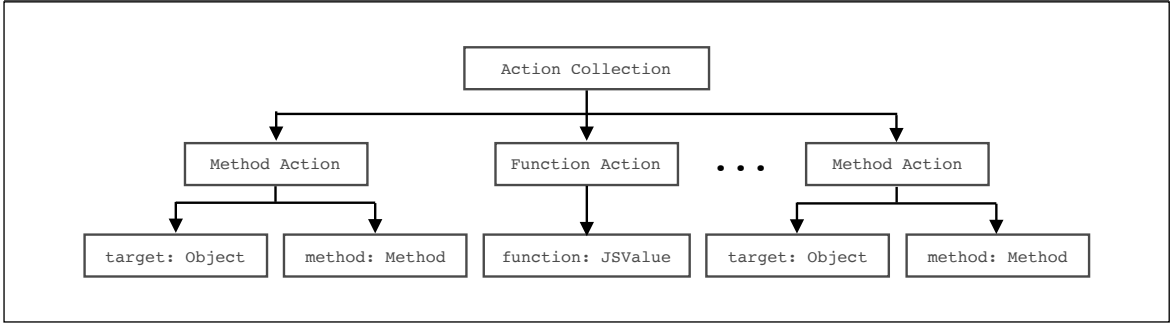
como um objeto que quer receber avisos sobre eventos, e então chamam um método específico que o gerenciador deve implementar, no qual ele recebe todas informações relevantes sobre aquele acontecimento (figura 8).

Isso significa que o gerenciador não possui um método só para lidar com a recepção de todos os eventos. Em vez disso, sua interface se adapta cada vez que uma interação nova é adicionada. Apesar disso, como o padrão adotado pelas bibliotecas é comum no sistema operacional (e acaba sendo comum em implementações de terceiros nesse ambiente), a adição de novas interações similares é simples, bastando se basear em outras conexões já existentes. Essa variedade também possibilita a adaptação de cada resposta ao evento a que ela se refere, permitindo a escolha de ações diferentes e a passagem de argumentos específicos.

Cada vez que o gerenciador é avisado sobre um acontecimento, ele procura em seus registros as ações que devem ser ativadas e as executa (figura 9). As ações são escolhidas pelo usuário e dependem de qualidades do evento recebido. Um evento pode ser global, como um gesto qualquer na tela; nesse caso, as ações globais correspondentes a gestos serão ativadas. Alternativamente, um evento pode estar relacionado especificamente a um item, como um gesto realizado no mesmo ponto da tela em que um item se encontra ou uma colisão entre dois itens na simulação de física. Nesse caso, o gerenciador usa um registro de ações contido no próprio item para achar e ativar as ações apropriadas.

As ações são objetos que encapsulam todas as informações necessárias para armazenar e executar pedaços de código. Elas normalmente contém funções implementadas em JavaScript, que são cidadãs de primeira classe e assim podem ser

FIGURA 9: Diagrama de objetos em UML, representando o registro de ações presente no gerenciador e em itens.
FONTE: Vendramini, 2015.



facilmente manipuladas. Alternativamente, usuários avançados podem usar ações para englobar métodos de Objective-C; nesse caso, o objeto guarda tanto o método a ser chamado quanto o objeto que receberá a chamada.

Ambas as implementações dependem de funcionalidades específicas das suas respectivas linguagens. Ações com código em JavaScript dependem não só de funções de primeira ordem, mas também da flexibilidade e facilidade de integração fornecidas pela biblioteca responsável (item 2.3.7). Já ações com código em Objective-C dependem de funcionalidades de meta-programação mais avançadas. Elas tomam vantagem da tipagem fraca e da resolução tardia de métodos da linguagem para armazenar os objetos e os métodos necessários. Assim, apesar da flexibilidade que trazem, elas não contam com a segurança normalmente fornecida pelo compilador e confiam que o programador deve saber o que faz – o que reforça o fato de serem uma funcionalidade voltada a usuários avançados.

A transmissão de parâmetros a esses pedaços de código é feita primariamente pelo gerenciador. Ele fornece às ações informações que só se pode saber em tempo de execução (como por exemplo a posição de um toque na tela). Caso a ação não precise desses dados, ela pode optar por passar ao código argumentos armazenados internamente. Esses argumentos podem tomar a forma de um vetor de parâmetros (que, devido à tipagem fraca, podem ser de diferentes classes e tipos) ou podem ser novas ações. Neste caso, forma-se um encadeamento arbitrariamente longo (mas finito) de ações, em que a última será avaliada primeiro, seu resultado será passado como argumento para penúltima, e assim por diante até que a cadeia termine.

Todos os formatos descritos foram utilizados para criar ações em algum ponto durante o desenvolvimento do projeto. A maior parte das ações resultantes está disponível para usuários finais. Essas ações "padrão" abstraem conceitos de computação gráfica e oferecem implementações para interações comuns. Entre elas estão os gestos de rotação, translação e escala usados na implementação do aplicativo (item 2.2.4).

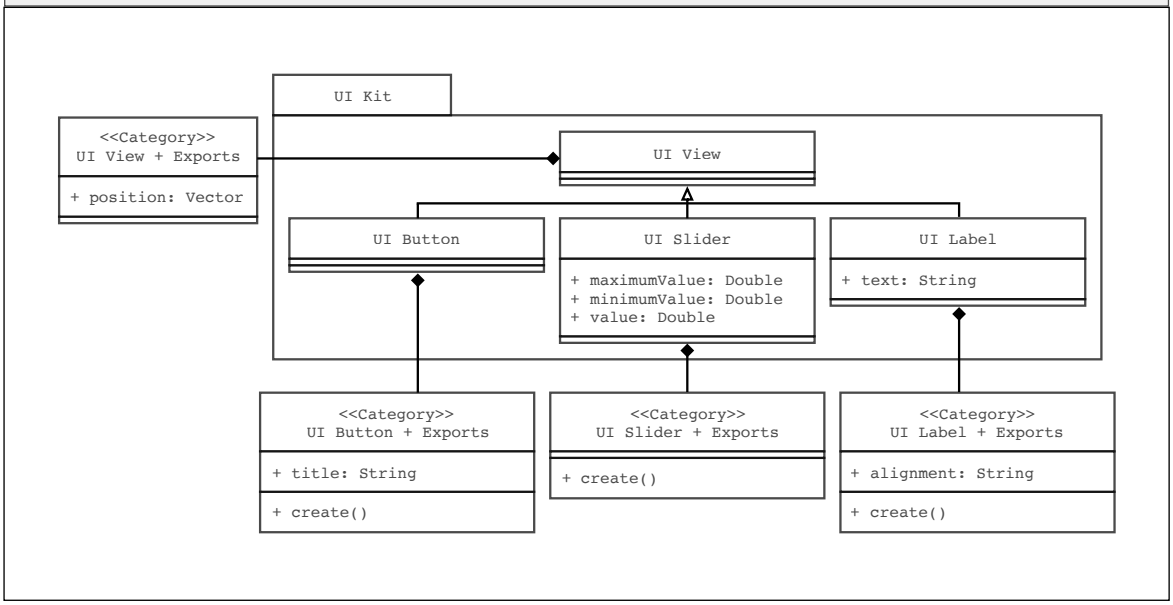
2.3.4 Interface de Usuário

Como o iOS é um sistema operacional que comporta uma grande quantidade de aplicativos funcionais (e não voltados exclusivamente para entretenimento), ao longo dos anos a implementação da sua biblioteca de interfaces (o principal componente usado na criação desses aplicativos) tem evoluído. Isso culmina atualmente em um conjunto de classes e métodos com grande potencial e flexibilidade, e que contém diversos recursos voltados para facilitar seu uso.

Essa biblioteca, chamada de UIKit, foi concebida para implementar funcionalidades que vão muito além do que se espera precisar em aplicativos que focam no conteúdo gráfico. Por causa disso, sua integração com o sistema foi feita de forma simplificada.

Em vez de se criar classes novas para empacotar uma implementação pré-existente (paradigma usado frequentemente no sistema), bastou a adição de uma série de métodos a essa biblioteca (figura 10). Esses métodos focam em simplificar ações comuns (como a criação de elementos de interface usando configurações padrão) e em tornar a interface da biblioteca mais similar a outras interfaces apresentadas pelo EngineKit. Assim, foi possível manter a implementação simples e leve e ainda assim adicionar todas as funcionalidades necessárias.

FIGURA 10: Diagrama de classes em UML, representando as categorias usadas para adicionar métodos ao UIKit.
FONTE: Vendramini, 2015.



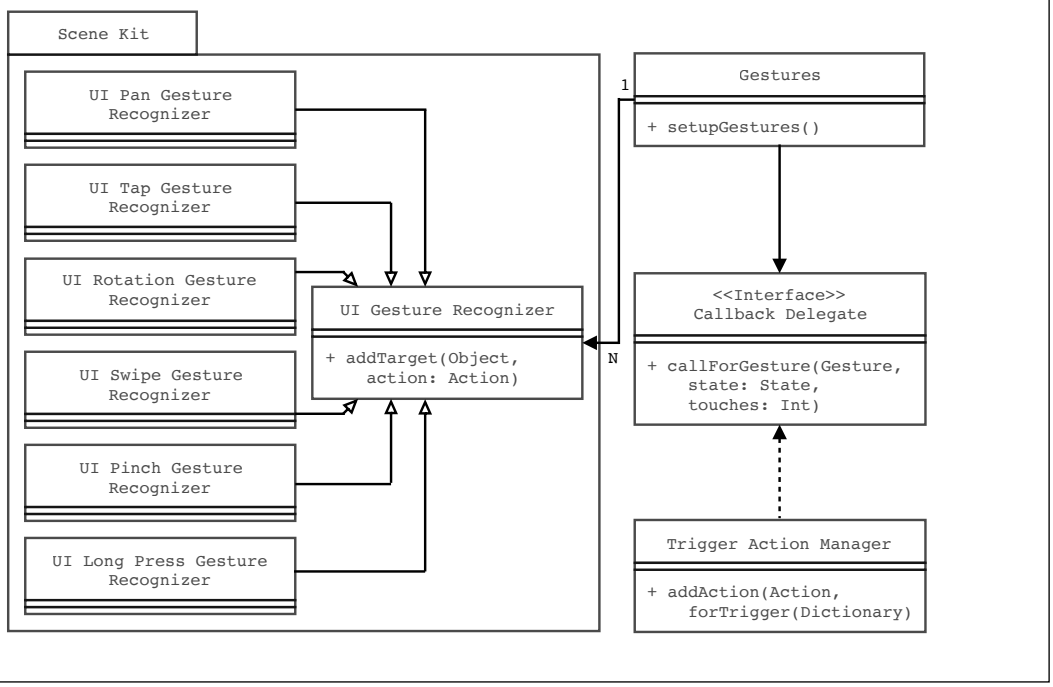
Como em outros subsistemas, caso um usuário avançado queira criar um conteúdo mais complexo (ou que envolva elementos ainda não integrados ao sistema), a implementação original da biblioteca também está disponível.

O UIKit é uma das bibliotecas responsáveis por detectar eventos e avisar o gerenciador de eventos e ações. Esses eventos representam interações do usuário com os elementos da interface, como por exemplo cliques em botões. A interface usada pela biblioteca para anunciá-los é similar a outras interfaces usadas pelo sistema operacional. Assim, ela se integra da mesma forma que as outras ao gerenciador de eventos e ações (item 2.3.3).

2.3.5 Gerenciador de Gestos

O subsistema que permite que o usuário de um aplicativo interaja com o EngineKit usando gestos na tela é a principal forma de interação da biblioteca. Alguns gestos específicos são usados nos elementos de interface comuns: botões, por exemplo, respondem automaticamente a toques. Apesar disso, o suporte a gestos variados e mais genéricos, realizados na cena em si (e não em um elemento específico), não é completamente coberto pelo UIKit e teve de ser alterado e adaptado.

FIGURA 11: Diagrama de classes em UML, a interação entre o subsistema gerenciador de gestos e o UIKit.
FONTE: Vendramini, 2015.



A biblioteca fornece um modo de identificar gestos que permite a sua separação em diferentes categorias (figura 11). Elas são identificadas de acordo com a intenção do usuário de (por exemplo) arrastar, girar, beliscar ou clicar na tela. Essa abordagem foi preferível à captação direta de informações não processadas pela biblioteca, já que ela agiliza o desenvolvimento e garante que gestos serão categorizados da mesma forma que no restante do sistema operacional.

A criação e configuração dos detectores de gestos é responsabilidade do gerenciador de gestos e é feita na sua inicialização. Cada detector é configurado para enviar as informações de cada gesto ao gerenciador. Esses gestos são então tratados: suas informações relevantes, como a posição na tela e o número de toques, são extraídas; e eles são analisados junto ao motor gráfico, para se saber quais objetos estão sob os dedos (e portanto quais são os alvos da interação).

Essas informações são então empacotadas e enviadas ao gerenciador de eventos e ações, que é responsável por transmiti-las às ações associadas. A interface entre o gerenciador de gestos e o de eventos e ações simula outras interfaces comuns do sistema operacional, garantindo a uniformidade citada anteriormente (item 2.3.3).

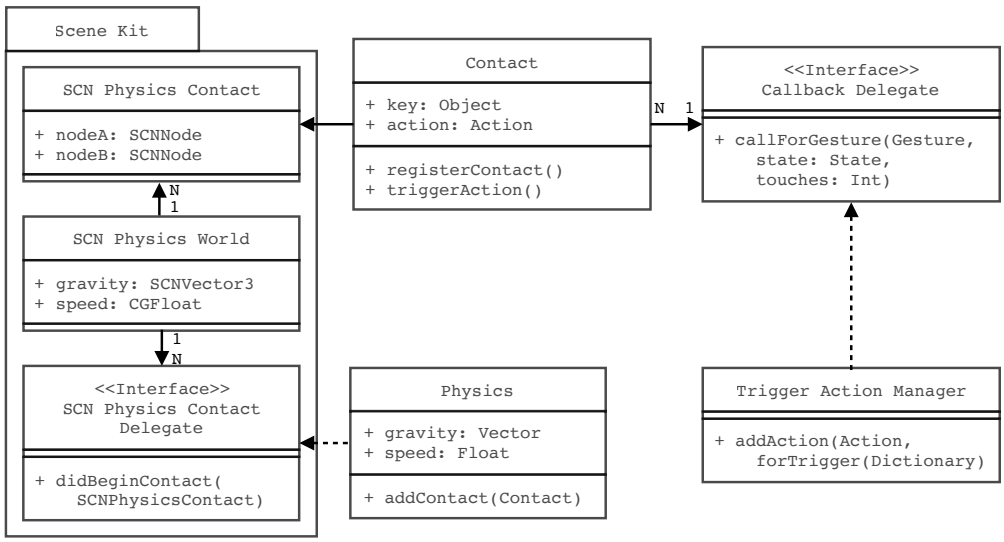
O gerenciador de gestos é responsável também por validar parâmetros e formatos apresentados durante a associação de eventos com ações. Como exemplo, em um gesto com a intenção de beliscar são necessários pelo menos dois dedos; uma requisição para associar uma ação a um gesto dessa categoria que seja executado apenas com um dedo é, portanto, inválida.

2.3.6 Simulador de Física

O SceneKit (motor gráfico usado pelo sistema) contém algumas funcionalidades para criar e manipular uma simulação de física na cena. O EngineKit usa essas funcionalidades no lugar de criar uma implementação própria, já que seu uso oferece as mesmas vantagens de praticidade, qualidade e performance que o de outras bibliotecas mencionadas anteriormente (figura 12).

Nessa simulação (que inicialmente está vazia), cada Item pode ser adicionado e então se comportar de formas variáveis, de acordo com a sua geometria. A simulação em si é

FIGURA 12: Diagrama de classes em UML, a interação entre o simulador de física e o SceneKit.
FONTE: Vendramini, 2015.



administrada pelo gerenciador de física, que apresenta formas de controlá-la como ajustes no vetor de gravidade e na velocidade de execução.

A simulação também é uma das fontes de eventos do sistema, que nesse caso correspondem a momentos de contato entre dois objetos. As respostas físicas a esse contato (como alterações das trajetórias e das velocidades dos objetos) são aplicadas automaticamente pela simulação. Após esse processo, as respostas do usuário são ativadas, permitindo executar ações não relacionadas ao processo físico (como mudanças estéticas ou alterações em outras partes da cena) ou sobrepor as mudanças automáticas da simulação, alterando manualmente propriedades como velocidade e posição dos objetos afetados pelo contato. O sistema resultante, portanto, permite que haja um controle fino sobre a simulação criada, possibilitando até comportamentos não-realistas (algo comum no desenvolvimento de jogos).

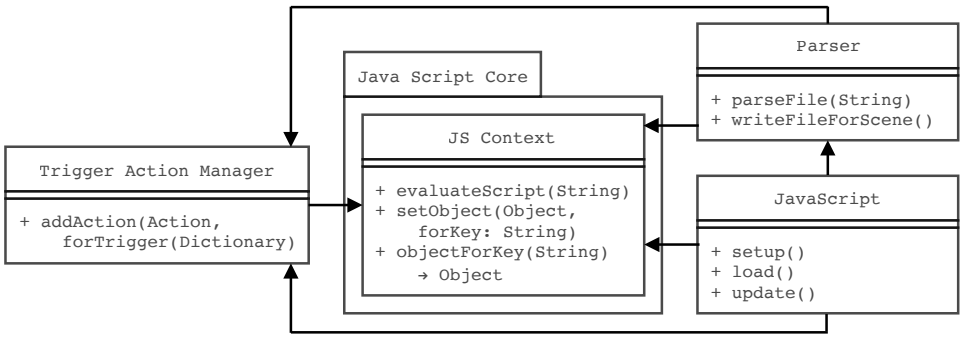
A administração de contatos é realizada por métodos da classe Contact. Essa classe cria uma correlação entre instâncias suas e de uma classe correspondente do simulador de física, chamada de SCNPhysicsContact. Diferentemente de outras classes do sistema, essa correlação não envolve um empacotamento de objetos. Em vez disso, cada contato detectado apresenta ao sistema uma instância diferente de SCNPhysicsContact, que então tem seu objeto Contact correspondente criado

dinamicamente. Este objeto então usa um conjunto de registros internos estáticos para obter suas informações relevantes (como os itens envolvidos no contato e as ações que devem ser ativadas) para completar suas tarefas.

Tanto o gerenciador de física quanto os objetos que representam eventos de contato são implementados como empacotamentos de funcionalidades da biblioteca utilizada, de forma que a simulação não precise lidar com elementos externos inesperados e possa permanecer estável mesmo após mudanças imprevistas pelo usuário.

2.3.7 Integração com JavaScript

FIGURA 13: Diagrama de classes em UML, representando interação entre os subsistemas que lidam com a linguagem *script* e o JavaScriptCore.
FONTE: Vendramini, 2015.



A integração do sistema com a linguagem JavaScript foi feita com o uso da biblioteca JavaScriptCore (figura 13), que permite uma integração natural com o Objective-C em diversos aspectos. É possível acessar e alterar valores em variáveis no JavaScript, exportar classes e funções escritas em Objective-C para que sejam usadas como *script* nativamente e criar múltiplas execuções de scripts diferentes paralelamente, para que sejam executados em cenas diferentes simultaneamente sem causar problemas de concorrência.

Os arquivos em JavaScript são lidos imediatamente após a configuração inicial da cena pela linguagem *markup*. Assim, o *script* tem acesso aos itens presentes na cena e aos modelos que foram declarados. É possível, portanto, executar sobre eles operações que vão além da capacidade da linguagem *markup* – seja por questões de

complexidade lógica ou porque dependem de fatores dinâmicos decididos em tempo de execução.

Entre essas operações, por exemplo, estão a adição de animações – que muitas vezes variam com o tempo e dependem de outros fatores (como eventos provocados pelo usuário) para acontecer; a criação de ações e sua associação a eventos (item 2.3.3); a adição e configuração de elementos de interface (item 2.3.4); e a implementação de algoritmos mais complexos, como inteligências artificiais.

O código em Java Script é lido uma vez no início do programa, o que significa que todo o código presente em escopo global será executado. Após essa primeira passagem, é possível recolher informações sobre as funções implementadas, que serão então chamadas conforme necessário. A execução de código em JavaScript funciona em um sistema de callbacks, em que funções são chamadas em pontos específicos do programa para responder a determinadas situações. Assim, o usuário não é responsável por tarefas básicas como a configuração e administração do laço de execução – esses pontos fundamentais são responsabilidade da linguagem nativa, o que garante confiabilidade, praticidade e eficiência.

As funções implementadas podem ser de três tipos. Uma função chamada *load*, caso exista, será executada assim que o sistema terminar sua configuração inicial, para que o usuário possa carregar as informações e os dados convenientes. Uma função chamada *update* será executada uma vez a cada frame, para permitir um controle fino da execução do programa (apesar de essa funcionalidade não ser recomendada por impactar negativamente a performance). Finalmente, funções que forem registradas como ações associadas a eventos (item 2.3.3) serão chamadas sempre que o evento em questão for detectado.

O sistema exporta todas as suas funcionalidades para o JavaScript, de forma a ser completamente acessível a seus usuários. Como as funcionalidades exportadas foram implementadas em linguagens nativas e frequentemente usam bibliotecas que melhoram sua performance, seu uso é preferível a implementações manuais na linguagem *script* dos mesmos algoritmos. Assim, a presença da linguagem visa

permitir a implementação de qualquer algoritmo que não seja suportado pelo sistema e, em particular, da lógica específica necessária para cada aplicativo.

2.3.8 Parser da Linguagem Markup

A classe Parser é responsável por administrar a conexão entre a linguagem *markup* de descrição de cenas e o EngineKit. Ela é capaz de traduzir informações de forma bidirecional, da sua representação em *markup* para estruturas de dado internas e vice-versa. Sua implementação está fortemente relacionada com a sintaxe da linguagem *markup*.

Essa sintaxe foi concebida com o objetivo de ter sua leitura e escrita acessível tanto para usuários quanto para programas. Para isso, ela mantém um conjunto de regras que aceita diversos modos de expressar o mesmo resultado. Essa decisão envolve sacrifícios na uniformidade do código: a variedade de opções compatíveis permite que desenvolvedores usem padrões consideravelmente diferentes entre si, tornando trabalhos alheios potencialmente mais difíceis de compreender. Apesar disso, como o sistema foi concebido visando a criação de aplicativos completos (e não de bibliotecas ou outras unidades de código dependente) é improvável que o compartilhamento de código seja um fator relevante o suficiente para seus usuários a ponto de valer a pena sacrificar a versatilidade obtida.

Essa amplitude de opções válidas permite que usuários escrevam seu código da forma que lhes for mais familiar. A figura 14 mostra exemplos de alternativas que seriam aceitas.

Outro fator diferencial com relação a outras linguagens *markup* populares envolve o fato de se ter evitado, sempre que possível, a adição de elementos que contribuíssem para a poluição visual do código. Como exemplo, a linguagem dispensa o uso de chaves, dois-pontos e outros caracteres comumente usados, dependendo exclusivamente de níveis de indentação para definir o escopo atual. Ainda assim, novamente com a intenção de apresentar uma sintaxe permissiva, o uso desses caracteres é suportado.

A flexibilidade da linguagem resultante torna o formato do arquivo de entrada imprevisível, e como é possível usar diversas opções dentro do mesmo arquivo, o sistema não se preocupa em simular os padrões fornecidos como entrada quando escreve arquivos de saída. Em vez disso, é usado um conjunto de padrões arbitrário, que visa (como antes) maximizar a legibilidade do resultado e dispensa estruturas e caracteres desnecessários.

Assim, mesmo que uma cena seja exportada imediatamente após ter sido importada, é provável que o arquivo de saída seja diferente do arquivo de entrada. Apesar disso, o sistema garante que ambos corresponderão à mesma cena, quando importados novamente.

2.3.9 Linguagem Markup

Os arquivos escritos com a linguagem *markup* são divididos em duas seções. A primeira, chamada de *templates*, permite que se crie modelos de Itens para se usar durante a construção da cena (figura 15). A segunda, chamada de *itens*, permite que esses modelos sejam instanciados, posicionados e configurados individualmente para montar a cena final (figura 16).

Itens declarados na primeira seção são inicialmente configurados de acordo com valores padrão. Cabe ao usuário associar às propriedades relevantes quaisquer valores que devam ser diferentes dos originais. As propriedades disponíveis incluem atributos válidos para qualquer Item, como posição, rotação e escala; atributos associados a geometrias físicas, como cor, massa e velocidade; e atributos relativos à geometria específica do Item em questão, como o raio de uma esfera ou as dimensões de um paralelepípedo.

FIGURA 14: Exemplo de código na linguagem *markup*.
FONTE: Vendramini, 2015.

```
color is red                  // com nomes
color is light gray           // com espaços
color is 0.08 0.24 0.27       // com números
color is (0.08, 0.24, 0.27)   // com separadores
color is [0.08, 0.24, 0.27];  // com ';'
color = {0.08 0.24 0.27};     // com '='
color is r:0.08 g:0.24 b:0.27 // com parâmetros
color is red = 0.08, blue = 0.24 and green = 0.27
```

A linguagem também permite que os modelos sejam montados de acordo com o sistema de hierarquias (item 2.3.2). Isso significa que modelos complexos podem ser criados a partir de combinações de outros modelos, desde que estes tenham sido declarados anteriormente ou sejam nativos do sistema. O parser usa níveis de indentação para percorrer a hierarquia e referencia sub-modelos por nome, de forma que cada declaração pode ter uma hierarquia arbitrariamente complexa, que pode ser percorrida para que propriedades sejam alteradas em qualquer profundidade.

Cada modelo criado é adicionado a um registro global da plataforma, de forma que cada declaração pode referenciar modelos criados em arquivos diferentes – desde que tenham sido lidos previamente.

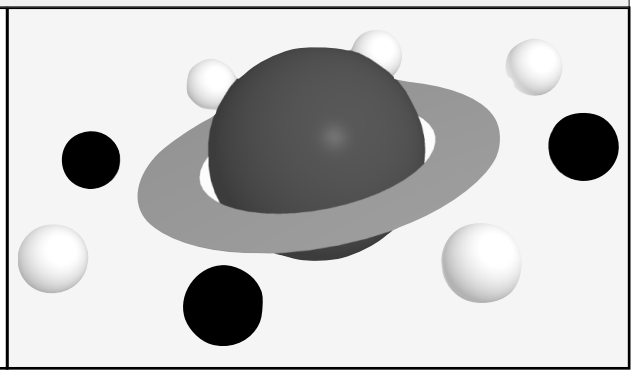
Na segunda seção, a instanciação de modelos é feita referenciando seu nome. Cada referência cria uma instância nova, que pode então ser configurada individualmente. Novamente, a sintaxe permite que propriedades em qualquer profundidade da

FIGURA 15: Exemplo de código na linguagem *markup*.
FONTE: Vendramini, 2015.

```
templates

    Moon Sphere
        color is white
        radius 0.1

    Planet Item
        Sphere PlanetBody
            color is darkGray
            radius is 1

        Tube Ring
            color is lightGray
            height is 0.01
            innerRadius is 1.1
            outerRadius is 1.7

        Item Moons

            Moon first
                position is 0.8 0 0
            Moon second
                position is -0.8 0 0
            Moon third
                position is 0 0 0.8
            Moon fourth
                position is 0 0 -0.8
            Moon fifth
                position is 0.6 0 0.6
            Moon sixth
                position is 0.6 0 -0.6
            Moon seventh
                position is -0.6 0 0.6
            Moon eighth
                position is -0.6 0 -0.6
```

FIGURA 16: Exemplo de código na linguagem *markup* e o modelo resultante da sua interpretação.
FONTE: Vendramini, 2015.

```
items

    Planet
        position is 0.2 0.3 0.4

        Moons
            scale is 3
            first
                color is black
            second
                color is black
            third
                color is black
```



hierarquia sejam alteradas. Neste caso, essas mudanças serão válidas apenas para a instância que está sendo configurada. Isso permite que haja variedade mesmo em Itens que compartilhem como base o mesmo modelo.

2.3.10 Biblioteca de Matemática

Para possibilitar que ambas as linguagens pudessem alcançar a flexibilidade desejada, foi necessário criar uma biblioteca de elementos matemáticos. Essa biblioteca é composta de classes que se baseiam em estruturas de dados compactas, compatíveis com o motor gráfico (mantendo assim toda a eficiência de *structs* em C), para representar elementos como vetores, rotações e ângulos. Com base nessas estruturas, foram implementados um conjunto de métodos para lidar com modos diversos de instanciação e com operações matemáticas comuns na computação gráfica.

Os métodos de instanciação são utilizados pelo parser da linguagem *markup* para permitir que propriedades sejam configuradas de diversas formas diferentes. Eles usam a tipagem condicional do Objective-C para determinar qual o tipo de objeto recebido como entrada e assim escolher o melhor método para extrair dele as informações necessárias. Assim, é possível criar elementos matemáticos a partir de representações em diversos tipos como números, vetores, strings e dicionários. Esta funcionalidade se torna particularmente útil quando exportada para JavaScript, já que a conversão automática de classes padrão torna o processo natural em ambas as linguagens (figura 17).

FIGURA 17: Exemplo de código em JavaScript, usando a biblioteca de matemática.
FONTE: Vendramini, 2015.

```
// |A + aB - C/2|

var A = vector.create([1, 2, 3]);
var B = vector.create({"x": 1, "y": 2, "z": 3});
var C = vector.create(1);
var a = 2;
var b = 3;

result = A.plus(B.times(a).minus(C.over(2)));


// (1x+2, 2y+4, 3z+6)

var S = Vector.create([1, 2, 3]);
var T = S.times(2);

result = A.translate(T).scale(S);
```

Os métodos de operações matemáticas se aproveitam do fato de a representação numérica interna ser baseada em tipos de C. Eles usam tanto operações quanto funções da biblioteca de matemática de C para garantir que essas operações serão executadas de forma eficiente. As operações implementadas vão desde somas e produtos simples até cálculos mais complexos de rotações ao redor de pontos arbitrários. Isso permite abstrair para o usuário parte das complicações matemáticas inerentes à computação gráfica.

As funções usadas para operações também foram projetadas de forma a melhorar a legibilidade das equações resultantes. A figura 17 mostra um exemplo do uso desses métodos em uma operação composta como alternativa à implementação manual.

3 Trabalhos Futuros

A meta estipulada para a biblioteca de se manter completa e flexível o suficiente para que se mantenha competitiva pode ter sido atingida, mas é necessário esforço contínuo em seu desenvolvimento ao longo do tempo para que tal qualidade se mantenha. Este esforço pode ser menor do que aquele necessário até agora – já que não envolve a implementação completa da ferramenta e sim os processos envolvidos na sua constante evolução e atualização – mas é essencial para a sobrevivência do sistema no mercado.

Não se pode prever quais atualizações serão necessárias ao longo do tempo, mas é certo que a evolução do programa e sua aproximação ao atual estado da arte tendem a aumentar sua competitividade. Apresentam-se então um conjunto de tarefas que podem servir como próximos passos e possíveis melhoramentos.

3.1 Linguagens

Foram escolhidas duas linguagens para a implementação do sistema em sua forma atual. Apesar de cada escolha ter sido feita ponderando um conjunto de alternativas, buscando aquela que apresentava mais vantagens, seria ingênuo ignorar o fato de que podem haver outras linguagens igualmente adequadas – ou então que trariam suas vantagens específicas ao sistema. Além disso, é provável que o suporte para diversas linguagens seja um fator atrativo para novos usuários, seja por verem a oportunidade de usarem uma linguagem conhecida ou por terem maior oportunidade de escolha.

Como o sistema foi desenvolvido como um conjunto de subsistemas relativamente independentes, a adição de novas linguagens poderia ser feita com a manutenção de grande parte da base de código.

Como exemplo, existem diversos aplicativos que contêm interpretadores de Python, Ruby e mesmo Lua (uma linguagem frequentemente usada na criação de jogos), todas possíveis alternativas ao JavaScript. Quanto à linguagem *markup*, a linguagem YAML apresenta uma sintaxe similar à usada pelo sistema, e a linguagem JSON apresenta recursos de tipagem e suporte nativo do iOS que a tornam uma opção interessante.

3.2 Recursos Gráficos

Como o EngineKit inclui um motor gráfico, o suporte a efeitos gráficos de diversos níveis é uma funcionalidade inerente. Seria interessante, portanto, expandir suas capacidades de renderização, manipulação de recursos e mesmo sua abertura a conteúdo criado externamente.

A biblioteca utilizada como motor gráfico, o SceneKit, oferece funcionalidades de alto nível adequadas para implementações desse tipo. Elas permitem o uso de sistemas de partículas, efeitos de iluminação e a utilização de recursos criados por usuários em outros programas, como malhas e objetos com formatos arbitrários, animações de esqueleto, e o uso de imagens e materiais como textura.

Felizmente, o sistema está bem equipado para incorporar tais mudanças. O encapsulamento fornecido pela estrutura de classes torna provável que as mudanças necessárias afetem pouco outros sistemas, enquanto as correlações de classes próprias com implementações do SceneKit faz com que as funcionalidades dessa biblioteca possam ser adicionadas naturalmente.

3.3 Funcionalidades do Sistema Operacional

O iOS é um sistema operacional que possui diversas bibliotecas, criadas para possibilitar o desenvolvimento de aplicativos com propósitos variados e abrangentes. Assim, existe uma série de adições que poderiam ser feitas ao programa apenas adicionando suporte a funcionalidades já implementadas.

A disponibilidade de elementos de interface para uso no programa, apesar de focar nos elementos mais utilizados, é consideravelmente limitada quando comparada à capacidade completa da biblioteca nativa. Apesar de ser improvável que elementos mais complexos (como tabelas e coleções) sejam necessários para muitos programas, o fato de estarem facilmente disponíveis é um incentivo à sua incorporação.

Existem também outros modos de interação como o usuário final que podem ser incorporados ao sistema. Diversos jogos fazem usos inovadores de acelerômetros e giroscópios desde os primeiros anos da plataforma. Além disso, tecnologias novas

chegam regularmente ao sistema operacional como, por exemplo, o toque com detecção de pressão, que também está sendo incorporado em jogos.

3.4 Acessibilidade

O sistema também tem por objetivo ser acessível a novos usuários. Para se aprofundar cada vez mais nesta meta e manter as vantagens que ela traz, é necessário que haja também esforços nesse sentido.

Para tanto, a realização de testes com possíveis usuários é indispensável. Apenas com o uso de dados obtidos de forma organizada a partir de interações reais será possível saber quais aspectos do sistema atingem a meta citada e quais precisam ser melhorados, além de se obter uma orientação sobre qual sentido esses melhoramentos devem seguir.

É importante também a criação de uma documentação abrangente do sistema implementado. Isso não só permite que usuários que queiram configurar a ferramenta para seus próprios fins possam fazê-lo de forma compreensível, mas também o torna mais convidativo a programadores que queiram participar do desenvolvimento do sistema.

4 Exemplos

Apresentam-se, nessa seção, exemplos de produtos criados usando o EngineKit. Eles foram concebidos para serem demonstrativos, não como produtos completos; sua presença visa demonstrar aspectos específicos do sistema sendo usados na prática. A não ser em casos em que se diga explicitamente o contrário, toda a implementação foi criada usando apenas as linguagem *markup* e *script*, sem manipular funcionalidades internas da biblioteca e sem acessar configurações avançadas. Os exemplos estão todos disponíveis no repositório do projeto.

4.1 O Planeta

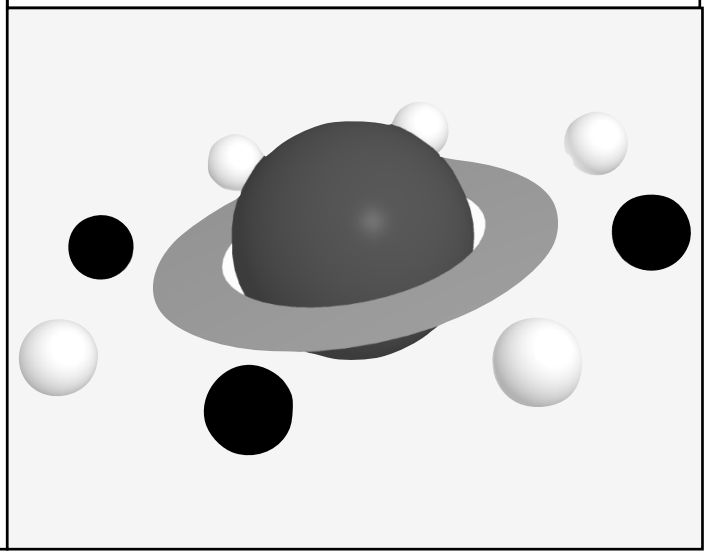
Os trechos de código em linguagem *markup* apresentados (figuras 15 e 16) podem ser unidos para criar um modelo que simule um planeta, com anéis e satélites (figura 18). Este exemplo usa o arquivo resultante dessa união para criar tal modelo, em conjunto com um arquivo *script* para fornecer controle sobre este modelo ao produto final.

O conteúdo do arquivo *markup* já foi discutido no item 2.3.9. Já no arquivo *script*, apresentam-se algumas funcionalidades novas, numeradas nos comentários da figura 18. A função de número 1 é implementada pelo sistema e adiciona à cena um conjunto de iluminação mais sofisticado que o oferecido por padrão pelo motor gráfico. A chamada de número 2 associa ao gesto de arrasto do dedo outra função implementada pelo sistema, que permite que o modelo seja girado. A chamada de número 3 associa ao gesto de clique a função implementada no número 4, que por sua vez aciona uma animação que faz o planeta começar a rodar em seu próprio eixo.

FIGURA 18: Exemplo de um par de arquivos, e seu modelo resultante. À esquerda o código escrito na linguagem *markup*. À direita, na linguagem *script*.
FONTE: Vendramini, 2015.

```
templates

    Moon Sphere
        color is white
        radius 0.1

    Planet Item
        Sphere PlanetBody
            color is darkGray
            radius is 1

        Tube Ring
            color is lightGray
            height is 0.01
            innerRadius is 1.1
            outerRadius is 1.7

        Item Moons

            Moon first
                position is 0.8 0 0
            Moon second
                position is -0.8 0 0
            Moon third
                position is 0 0 0.8
            Moon fourth
                position is 0 0 -0.8
            Moon fifth
                position is 0.6 0 0.6
            Moon sixth
                position is 0.6 0
-0.6
            Moon seventh
                position is -0.6 0
0.6
            Moon eighth
                position is -0.6 0
-0.6

items

    Planet
        position is 0.2 0.3 0.4
        rotation is 0 1 0 0

        Moons
            scale is 3

            first
                color is black
```

```
function load() {
    // 1
    loadStandardLights();

    // 2
    TriggerManager.addActionForTrigger(trackballAction,
                                       {"gesture": "pan"});

    // 3
    TriggerManager.addActionForTrigger(handleTap,
                                       {"gesture": "tap"});
}

// 4
function handleTap() {
    planet.addAnimation({"keyPath": "rotation.w",
                         "toValue": 2 * pi,
                         "repeatCount": 99,
                         "duration": 5});
}
```

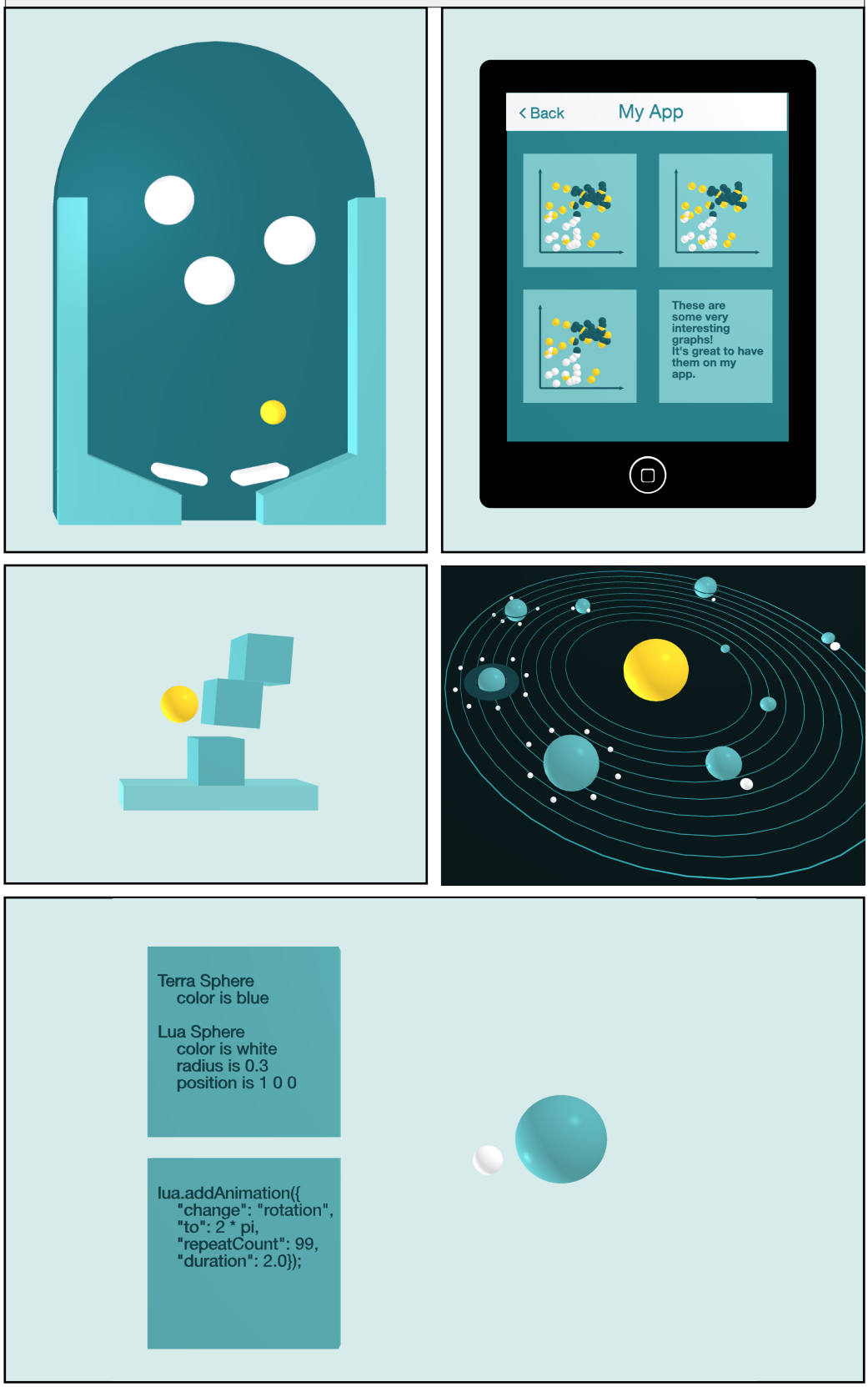


4.2 A Apresentação

Como parte do desenvolvimento do projeto, teve de ser criada uma apresentação do sistema, incluindo detalhes sobre seu propósito e suas funcionalidades. Esta apresentação foi feita na forma de um vídeo, que é na verdade uma narração gravada sobre uma captura de tela, exibindo um aplicativo implementado usando o EngineKit.

Esse aplicativo foi desenvolvido com propósito de servir como acompanhamento para uma apresentação qualquer, seja ela em vídeo ou presencial. Assim, ele conta com ferramentas para a interação e o controle de seu andamento em tempo real. O

FIGURA 19: Exemplos retirados do vídeo de apresentação. Do canto superior esquerdo, no sentido horário: uma simulação física de um jogo de pinball, um modelo de aplicativo, um modelo animado do sistema solar, um exemplo de arquivos e seu modelo resultante e uma simulação de colisão entre uma esfera e uma torre de blocos.
FONTE: Vendramini, 2015.



programa se baseia em apresentações convencionais, exibindo uma série de modelos, símbolos e deixas textuais em sequência para ilustrar o que é dito pelo apresentador. Mas, além disso, ele faz uso de algumas funcionalidades particulares do EngineKit — como animações, efeitos de iluminação e simulações de física — para tornar a apresentação mais dinâmica, ilustrar melhor o que está sendo dito e exibir as diferentes funcionalidades que a biblioteca é capaz de oferecer.

Algumas capturas de tela da apresentação são exibidas para demonstrar alguns modelos mais complexos usados e tentar exemplificar (apesar da falta de movimento) as animações e simulações presentes.

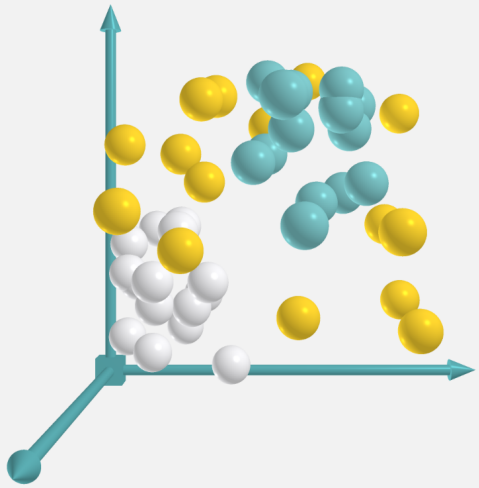
4.3 Criação em tempo de execução

Uma das vantagens do paradigma de linguagens apresentado é o fato de o conteúdo ser completamente baseado em linguagens interpretadas em tempo de execução. Isso significa que não é necessário possuir quaisquer informações sobre o conteúdo em tempo de compilação, o que abre duas possibilidades interessantes: a habilidade de se decidir o que será exibido na tela com base em informações calculadas na mesma hora; e a capacidade de interpretar informações obtidas de fontes independentes, quer se originem de outros pontos no mesmo aplicativo, quer tenham origens externas — como outros aplicativos, outros dispositivos ou mesmo servidores acessados pela internet.

Esse fato abre para programadores diferentes possibilidades. Como exemplo, apresenta-se um aplicativo que usa configurações um pouco mais avançadas que os anteriores. Esse programa usa um *script* em Perl que explora a simplicidade da sintaxe da linguagem *markup*, criando programaticamente um arquivo nessa linguagem. O *script*, que neste contexto representa uma "fonte externa", modela eixos de um gráfico em três dimensões e esferas para representar pontos nesse gráfico, e então posiciona as esferas de acordo com pontos fornecidos pelo gerador de números pseudo-aleatórios nativo da linguagem. O arquivo então é lido pelo programa e exibido na tela, resultando na representação abaixo.

FIGURA 20: Um *script* em Perl (à esquerda) que cria um arquivo na linguagem *markup*, e o modelo (à direita) gerado por esse arquivo.
FONTE: Vendramini, 2015.

```
print "templates\
    Axis Item\
        color is 0.35 0.67 0.69\
        Cylinder line\
            height is 4\
            radius is 0.06\
            position is 0 1.94 0\
        Cone point\
            height is 0.3\
            radius is 0.12\
            position is 0 4 0\n\
    DataPoint Sphere\
        radius is 0.2\n\
    DataPointA DataPoint\
        color is 0.49 0.78 0.79\n\
    DataPointB DataPoint\
        color is 0.95 0.95 0.96\n\
    DataPointC DataPoint\
        color is 0.99 0.83 0.18\n\
    Graph3D Item\
        Axis\
        Axis\
            rotation is 0 0 1 -1.5707\
        Axis\
            rotation is 1 0 0 1.5707\
        Box\
            width is 0.3\
            height is 0.3\
            length is 0.3\
            color is 0.35 0.67 0.69\n\
items\
    Graph3D\n\n";

for ($i = 0; $i < 15; $i++) {
    $x = rand() * 1.5 + 1.5;
    $y = rand() * 1.5 + 1.5;
    $z = rand() * 1.5 + 1.5;
    print "    DataPointA\
        position is $x $y $z\n";
}
for ($i = 0; $i < 15; $i++) {
    $x = rand() * 1.5 + 0.2;
    $y = rand() * 1.5 + 0.2;
    $z = rand() * 1.5 + 0.2;
    print "    DataPointB\
        position is $x $y $z\n";
}
for ($i = 0; $i < 15; $i++) {
    $x = rand() * 3 + 0.2;
    $y = rand() * 3 + 0.2;
    $z = rand() * 3 + 0.2;
    print "    DataPointC\
        position is $x $y $z\n";
}
```



5 Conclusão

Existe, atualmente, uma quantidade crescente de desenvolvedores inexperientes começando seu aprendizado em técnicas e ferramentas da computação, visando tomar proveito de oportunidades cada vez mais numerosas no mercado recém-criado de aplicativos móveis. Em particular, a área de computação gráfica tem se expandido durante as últimas décadas, de forma que sua presença neste mercado hoje é significativa. Assim, faz-se necessária a presença de ferramentas que sejam voltadas para este público.

Neste projeto foi desenvolvida uma tal ferramenta, que atinge seu objetivo de auxiliar no desenvolvimento de conteúdo gráfico em plataformas móveis, priorizando a acessibilidade e mantendo-se completa o suficiente para servir como alternativa a programas similares. Apesar de o programa apresentado cumprir suas metas, há ainda espaço para melhoramentos e atualizações ao longo do tempo para que ele evolua e se mantenha competitivo. Assim, pretende-se continuar a trabalhar com o sistema, mesmo após sua publicação.

Durante o desenvolvimento do projeto, diversos desafios foram encontrados, muitos dos quais só puderam ser resolvidos com métodos aprendidos durante o curso. Para dificuldades técnicas, o estudo de computação gráfica foi essencial. Para nortear planejamentos iniciais, a engenharia de software foi particularmente útil. Para possibilitar adaptações de metodologia, garantir a agilidade e a adequação a mudanças durante o processo, ajudar a espreitar visões mais amplas de como o sistema completo deveria se comportar e manter o foco em decisões de maior importância, os estudos no laboratório de programação extrema foram indispensáveis.

Estes desafios poderiam apenas ser enfrentados durante a implementação de um sistema completo, e todos acabaram fornecendo suas próprias contribuições para o aprendizado resultante deste projeto. Este aprendizado inclui um novo foco no usuário e novas formas de decidir quais funcionalidades são de fato importantes; a versatilidade que permite adaptar abordagens frente a um problema, por maior que

seja a mudança resultante; e a importância de motivações em um projeto de maior escala, que impulsionam e direcionam seu desenvolvimento.

Referências

GREGORY, Jason. Game Engine Architecture. Boca Raton: CRC Press, 2015.

POCKETGAMER. **App Store Metrics**, 2015. Disponível em <http://www.pocketgamer.biz/metrics/app-store/>. Acesso em 30/11/2015.

STATISTA. **Percentage of app developers targeting connected devices as of 1st quarter 2014, by type**, 2014. Disponível em <http://www.statista.com/statistics/256570/devices-targeted-by-app-developers/>. Acesso em 30/11/2015.