



GAME 3D



ÍNDICE



Introdução

4

MERCADO DE GAMES	5
PROFISSÕES	6
O JOGO	7
CRONOGRAMA DO CURSO	8



Modelagem

9

INTERFACE E COMANDOS BÁSICOS	11
POSICIONAMENTO	17
DUPLICATAS	19
PARENTESCOS	20
ORGANIZAÇÃO	22
FERRAMENTAS DE MODELAGEM	24
MATERIAIS E MAPA UV	32
MODELAGEM ORGÂNICA	38

ÍNDICE

Animação

42



ANIMAÇÃO	44
INTERFACE DE ANIMAÇÃO	45
CURVAS DE ANIMAÇÃO	47
BLEND SHAPE	50
RIGGING	51
PRINCÍPIOS DE ANIMAÇÃO	59

Texturização

61



INTRODUÇÃO	63
INTERFACE E COMANDOS BÁSICOS	64
ATRIBUTOS BÁSICOS DE MATERIAIS	67
TOOLS	70
CAMADAS	72
INICIANDO O PROJETO	73
BAKE MESH MAPS	75
MÁSCARAS	76
SMART MATERIALS	77
ADD EFFECTS	77
EXPORTAÇÃO DE TEXTURAS	78

ÍNDICE



INTRODUÇÃO	80
UNITY 3D	81
INTRODUÇÃO A PROGRAMAÇÃO	81
FLAPPY BIRD	87
MECÂNICAS DE JOGOS	100
TPSHOOTER	102
UNITY 3D - 2	114



INTRODUÇÃO

MERCADO DE GAMES

PROFISSÕES

O JOGO

CRONOGRAMA DO CURSO

PLAY

INTRODUÇÃO

Nós da MK Academy com muito prazer damos boas vindas ao curso de Game 3D. Nos próximos meses vamos explorar as ferramentas e conceitos necessários para produção de jogos 3D, sendo um curso onde se aprende na prática e já se sai com jogos produzidos, um primeiro passo para ingressar profissionalmente nesse mercado em franca expansão no Brasil e no mundo.

Nada mais justo para iniciar esta jornada do que entender um pouco mais deste universo gamer, questões sobre o mercado, seu potencial e particularidades, como se dá o fluxo na produção de jogos e funcionamento geral de nosso curso como um todo. Este módulo tem o objetivo de introduzir estes conceitos apresentando um panorama geral.

MERCADO DE GAMES

Não é segredo que o mercado de entretenimento como um todo costuma movimentar bastante a economia pelo globo, setores como eventos, música, cinema, geram receitas impressionantes, e o mercado de jogos digitais nos últimos anos tem conseguido desbancar esses gigantes, sendo um dos setores do entretenimento que mais geram receita. De jogos simples jogados nos minutos ociosos de uma espera aos profissionais do E-sports, os jogos fazem parte da vida das pessoas e girar as engrenagens dessa máquina requer muitos profissionais, cada vez mais preparados.

1. MERCADO DE JOGOS NO BRASIL

Obviamente o mercado consumidor brasileiro não é uma exceção a esta tendência, sendo um dos maiores do mundo, por vezes o maior da América Latina, a grande maioria da população, entre jovens e adultos, consomem algum tipo de jogo digital. Como desenvolvedores também temos relevância, não apenas como mão de obra em empresas pelo mundo, como no grande número de empresas nacionais, localizadas em todas suas regiões, que alimentam o mercado interno e o mundial com produtos de qualidade. Todos os anos temos eventos em que os desenvolvedores mostrar seus produtos como **Brasil Game Show**, Os acadêmicos mostram suas pesquisas e tecnologia e impacto social dos jogos como a **SBGames (Simpósio Brasileiro de Games)**, ainda a os concursos governamentais como o do **SEBRAE** e os editais da **ancine** que ajudam a fomentar a produção nacional, Resumindo a Profissão de desenvolvedores de jogos já tem o reconhecimento como sério em nosso país.

2. ACESSO PROFISSIONAL

O ato de ingressar profissionalmente no mercado de jogos tem ficado cada vez mais acessível, o que no passado parecia um sonho distante, tem se mostrado uma realidade por uma série de fatores. Como dito no tópico anterior existem várias empresas nacionais e estas precisam de mão de obra, e com as facilidades do trabalho remoto não é inviável trabalhar daqui para empresas de fora, além disto publicar jogos por conta própria tem se mostrado um processo cada vez mais fácil, com plataformas como a steam e as de jogos mobiles, não é uma tarefa impossível formar equipes e publicar seus jogos, e as grandes empresas com a desobrigatoriedade de impressão de mídia física perceberam que é um bom negócio abrir as portas para os indies. Conhecendo os canais e tendo qualidade nos produtos desenvolvidos as coisas podem acontecer.

PROFISSÕES

Profissionais reconhecidos, os desenvolvedores de jogos tem diversas funções em um projeto, Seja em produções independentes onde as funções se acumulam ou em grande empresas que oferecem cargos de forma mais especializada, existem várias ramificações dentro desta profissão. Obviamente todos têm uma noção básica do processo como um todo.

1. ART DESIGN

Os “artistas”, geralmente cuidam da parte gráfica do jogo, personagens, cenários, efeitos, ilustrações, etc, o que envolve diversas áreas como animação, partículas, character design, texturização, concept, entre vários outros, além das diferenças de recursos feitos para jogos 2D ou 3D. Conhecimentos ligados a estética como teoria cromática, linguagem visual, anatomia, são alguns dos recursos que podem ser úteis de acordo com sua especialidade dentro deste ramo.

2. PROGRAMADOR

São os responsáveis pela parte de implementação do jogo, conectar os comandos dos jogadores com as ações no jogo e fazer com que funcione cada elemento dentro do jogo, com suas características e meio de interação entre eles. Linguagens de programação, lógica de programação, matemática, são ferramentas de um programador.

3. GAME DESIGN

Game designer são os profissionais responsáveis pela criação das “regras” do jogos, desenvolvem interações, conectam enredo e mecânicas e se certificam de fazer o jogo ser divertido. design de interação, level design, user experience, são algumas das ramificações. ter uma noção geral do processo de desenvolvimento do jogo ajuda na tomada de decisões.

4. AUDIO DESIGN

Criam os elementos sonoros dos jogos, geralmente trabalham com trilhas e efeitos, além dos conhecimentos óbvios ligados a música o audio designer ter que se preocupar com coisas como interatividade e como os diversos elementos criados vão funcionar juntos em tela.

5. PROFISSÕES RELACIONADAS

Vários outras profissões podem tangenciar a indústria de games, roteiristas, designers de interface, publicitários, pesquisadores, testers, críticos, professores e como em todos mercados há todos estes profissionais que orbitam a produção de jogos.

O JOGO

Que os games tem impacto econômico e social é um fato, o que nos faz gostar deles e o que os definem? que elementos os compõem? Um desenvolvedor tem que ter noções básicas das respostas destas questões. Compreender o que é um jogo e o que fazem eles atrativos ajuda a fazer escolhas conscientes no design.

1. O QUE É UM JOGO

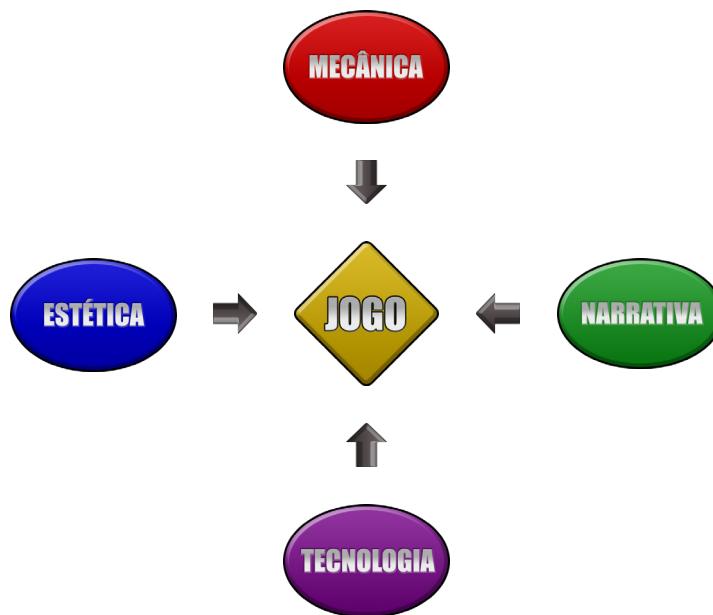
Definir o que é um jogo parece uma tarefa fácil, no entanto existem tantos tipos bem distintos entre si, de brincadeira de faz de conta de crianças e jogos olímpicos, passando pelo console em sua casa, todos são jogos. Se olharmos no dicionário veremos algo como “ato de jogar, divertimento, brincadeira”, o jogo está atrelado ao jogar, se não a **interação** não é jogo, e o jogo normalmente tem a função de **divertimento**.

2. O QUE FAZ UM JOGO DIVERTIDO

Se um jogo é “uma interação para divertimento”, o que faz com que esta interação seja divertida? Quando jogamos normalmente enfrentamos desafios e experimentamos situações, sem que nos arrisquemos ou percais algo drástico em nossas vidas, entramos em um conflito artificial onde podemos experimentar a vitória e as vivências de forma segura. Os estudiosos chamam esta sensação de **círculo mágico** uma imersão prazerosa que torna os jogos tão atrativos.

3. O QUE COMPÕE UM JOGO

Quando pensamos em um jogo, muitas vezes o seu enredo vem a mente, por outras vem seu gênero (battle royal, RPG, puzzle, etc), no entanto uma história pode gerar jogos bem diferentes, assim como um gênero pode proporcionar diferentes experiências de acordo com como seu aspecto é tratado. Um jogo normalmente tem alguns elementos que juntos ditam a experiência do jogador. Segundo a **tétrade de Schell** um jogo tem quatro pilares principais: **narrativa** (a história e os fatos que se desenrolam), **estética** (o audiovisual a “cara” do jogo), **tecnologia** (onde o jogo rola e que recursos ele usa) e **mecânica** (a jogabilidade, as “regras” do jogo). A junção destes elementos de forma harmônica dão a identidade do jogo.



CRONOGRAMA DO CURSO

Nosso curso é dividido em 8 módulos desenvolvidos de forma a seguir um fluxo de produção de recursos e desenvolvimento de um jogo, estes módulos têm uma duração média (varia de acordo com os feriados e recessos) de 18 meses.

MÓDULO	DURAÇÃO (Meses)	SOFTWARE
Introdução ao Universo do Gamer	1	Maya / Photoshop / Unity
Modelagem	3	Maya
Texturização	2	Substance Painter
Animação	1	Maya
Rigging	1	Maya
Reforço de conteúdo	1	Conteúdo livre no tema: Game
Unity	4	Unity
Produção do jogo	5	Maya / Substance Painter / Photoshop / Unity

1. INTRODUÇÃO AO UNIVERSO DO GAMER

Contextualização ao desenvolvimento de jogos e ao curso como um todo, fase de discussão e análise de conceitos que cercam o universo gamer e na apresentação das ferramentas que serão utilizadas durante o curso.

2. MODELAGEM

Focado na criação de modelos 3D no autodesk Maya, mas especificamente na criação dos modelos de objetos, cenários e personagens, levando em conta o necessário para o seu bom funcionamento em um jogo.

3. TEXTURIZAÇÃO

Criação de texturas utilizando o Substance painter, basicamente a criação de mapas que determinam cores, brilhos, reflexão entre outros atributos de um modelo, fazendo uma pintura direta sobre os modelos 3D.

4. RIGGING/ANIMAÇÃO

Trata de como preparar os modelos para animação, criando esqueletos que os articulam e estudar conceitos de como fazer animações de qualidade

5. REFORÇO DE CONTEÚDO

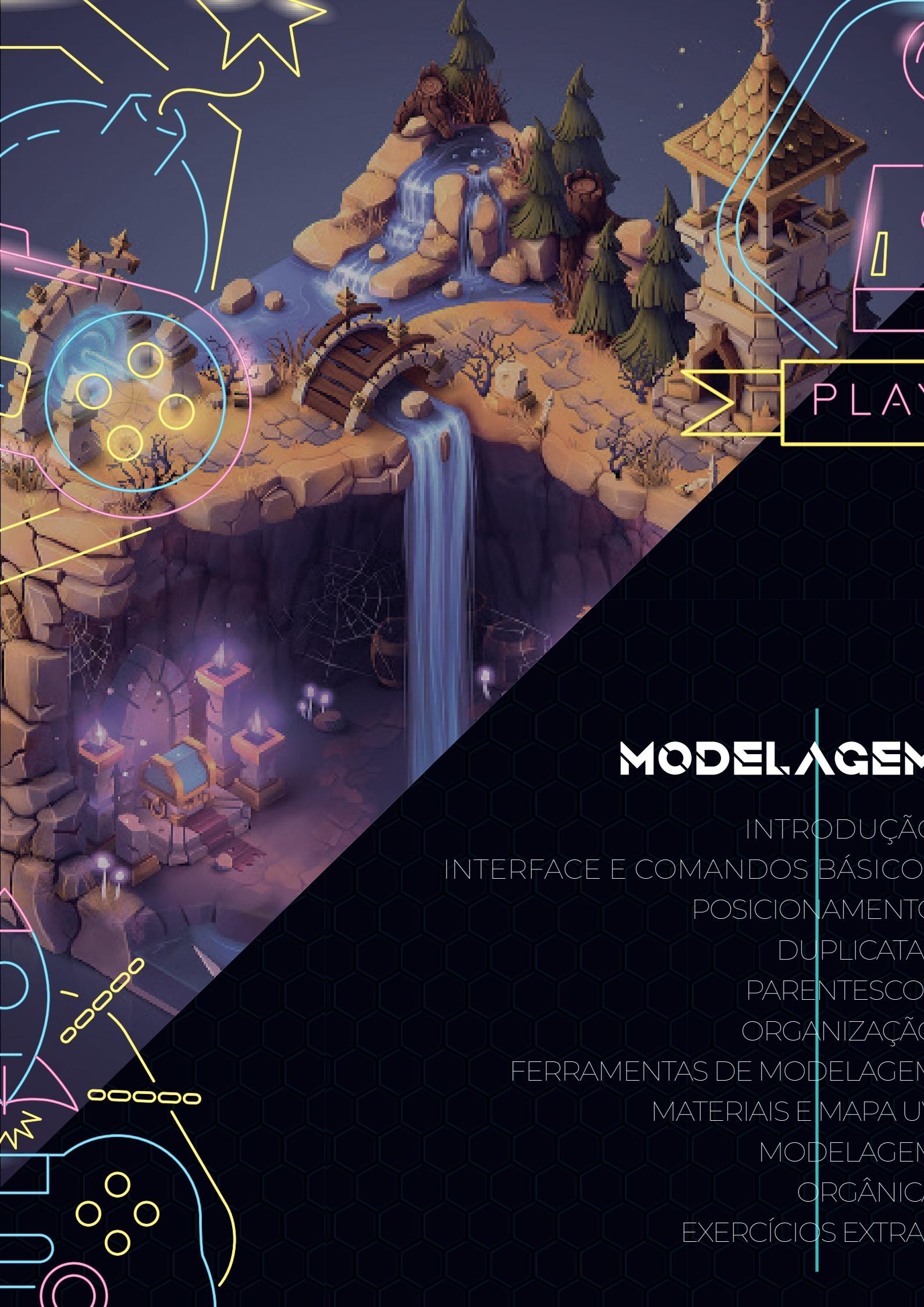
Este módulo está destinado a suprir demandas da turma, como reforço de algum outro módulo ou acrescentar conteúdo que complementa a formação.

6. UNITY

É o módulo que aborda a criação das funcionalidades do jogo, onde haverá a junção dos elementos que compõem o jogo, dentro do Unity.

7. PRODUÇÃO DE JOGO

Conclusão do curso, onde os alunos vão fazer seus jogos sob a orientação dos professores.



MODELAGEM

INTRODUÇÃO
INTERFACE E COMANDOS BÁSICO
POSICIONAMENTO

DUPLICATA

PARENTESCO

ORGANIZAÇÃO

FERRAMENTAS DE MODELAGEM

MATERIAIS E MAPA U

MODELAGEM

ORGÂNICA

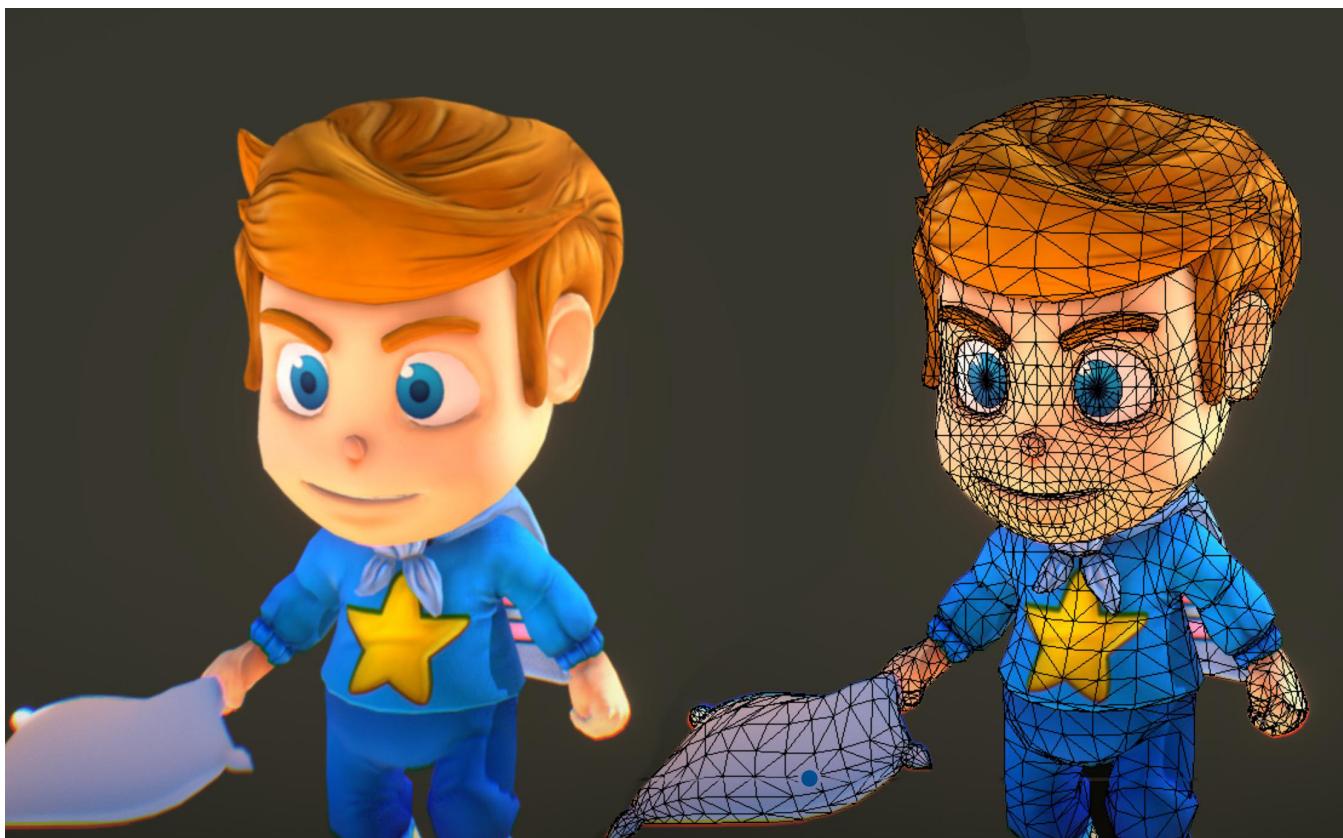
EXERCÍCIOS EXTRA

INTRODUÇÃO

Modelos 3D estão presentes nas mais diversas áreas de trabalho na atualidade, tanto na criação de jogos e filmes, que são mercados de importância inquestionável por seu destaque na economia, como por seu impacto na cultura, também estão presentes no planejamento e prototipação de mecanismos simples a edificações, apresentando visualmente conceitos de objetos e espaços variados, deixando o processo de produção mais rápido e barato. Também é importante notar que com tecnologias como a impressão 3D e dispositivos de Realidade virtual o leque de oportunidades profissionais de um modelador 3D se amplia de forma significativa, sendo este um profissional cada vez mais requisitado.

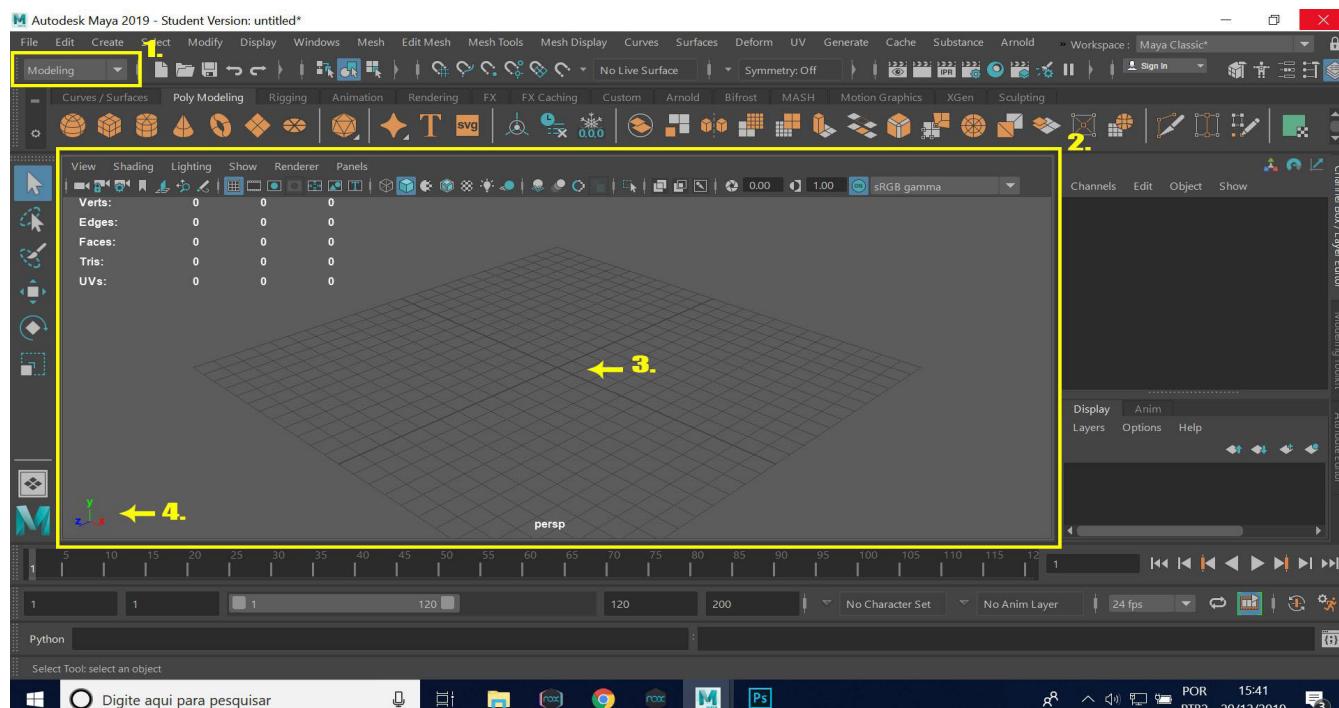
Mas afinal de contas o que é e como se cria uma modelagem 3D? Um Modelo 3D nada mais é do que uma representação em ambiente virtual de um objeto com proporções calculadas em três dimensões: altura, largura e profundidade. Estes objetos são construídos de vários pontos em que o posicionamento de cada um deles no espaço virtual corresponde a uma coordenada, sendo assim possível determinar quando algo está acima, ao lado ou a frente de outro elemento, gerando assim um espaço que em alguns aspectos é semelhante ao mundo em que vivemos. Essa semelhança torna interações com estes espaços muito mais imersivas, capazes de nos levar a mundos inexplorados de aventura e mistério.

Neste módulo trabalharemos com o Autodesk Maya, um software de alto padrão, um dos softwares líderes de mercado, utilizado em grandes produções consagradas de diversos segmentos, sendo uma ferramenta poderosa tanto para modelagem, efeitos e animação em ambiente 3D, estando o foco nesta primeira etapa na produção de objetos e personagens tridimensionais.

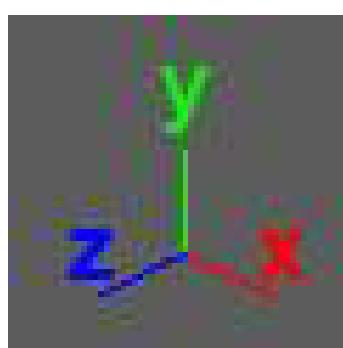


INTERFACE E COMANDOS BÁSICOS

A princípio a interface do Autodesk Maya pode parecer complexa, no entanto é importante ter em mente que este é um programa capaz de executar tarefas bem distintas como: modelagem, animação, física, partículas, é não é preciso aprender todos estes de uma só vez, sendo possível setorizar seu aprendizado, além disto este software apresenta diversas ferramentas que conseguem atingir resultados semelhantes possibilitando ao modelador escolher seu próprio fluxo de trabalho.



01. Esta caixa permite alterar a interface de acordo com a tarefa que está sendo executada, animação, *rigging*, etc. Por hora mantenha *modeling* selecionada.
02. Esta é nossa *viewport*, basicamente nossa área de trabalho que simula nosso ambiente 3D.
03. Grid, esta grade nos dá a noção de um piso para nos orientar quanto ao posicionamento de um determinado objeto no mundo.
04. *Gizmo*, este pequeno ícone nos ajuda na orientação dos eixos em relação a câmera, as três letras representam as dimensões, sendo **Y** altura, representado pela cor verde, **X** largura, representado pela cor vermelha, e **Z** profundidade, representado pela cor azul. Estes eixos sempre se modificam para se ajustar à rotação da câmera.

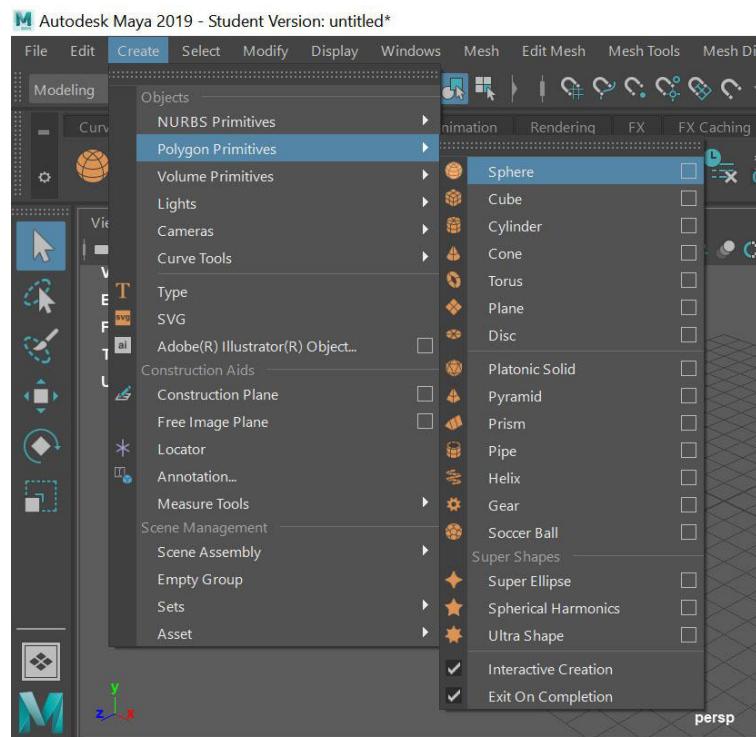


Gizmo.

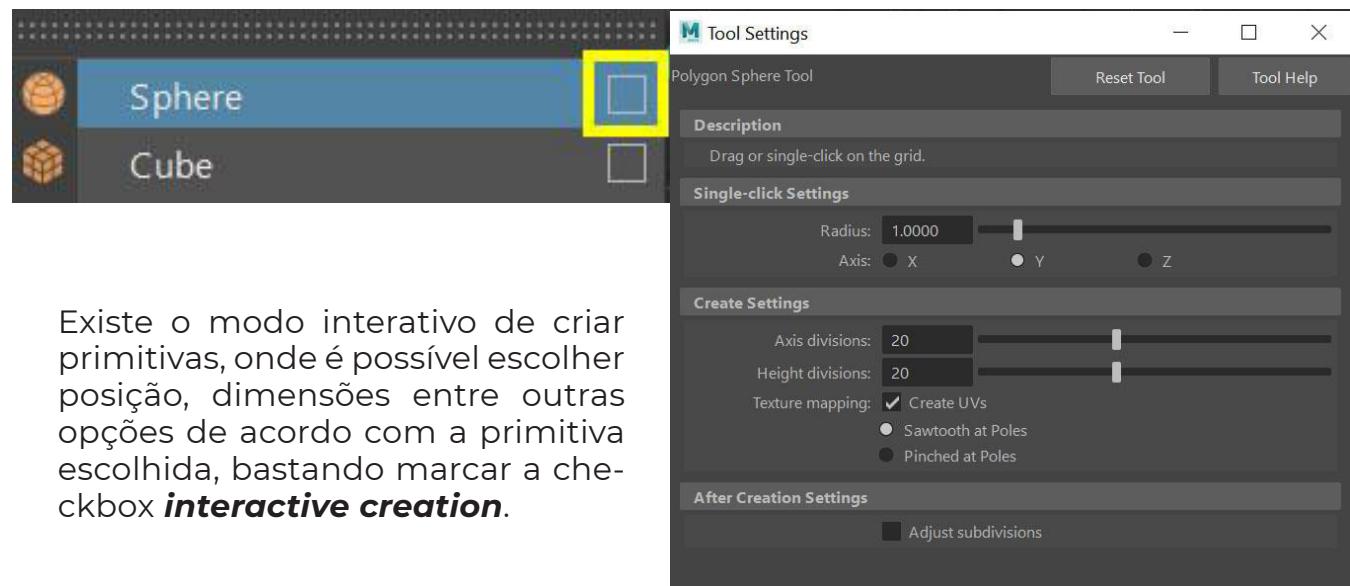
1. CRIANDO PRIMITIVAS

Existem formas pré-construídas, as primitivas, são formas básicas que o software disponibiliza para construção rápida, é comum utilizar estas formas como uma base inicial para um trabalho mais complexo.

Create>Polygon Primitives



Por padrão quando uma primitiva é criada ela aparece em sua configuração básica, posicionada no centro do mundo (ponto 0,0,0). Para alterar os parâmetros de criação do objeto basta clicar no quadrado ao lado da opção, sempre que este quadrado aparecer significa que a opção em questão tem um menu de configuração.



Existe o modo interativo de criar primitivas, onde é possível escolher posição, dimensões entre outras opções de acordo com a primitiva escolhida, bastando marcar a checkbox **interactive creation**.

Create>Polygon Primitives>Interactive Creation

2. MOVIMENTAÇÃO DE CÂMERA.

Para visualizarmos nossos modelos 3D é necessário aprender a manipular nossa câmera utilizando os seguintes comandos:

» Girar a câmera:



» Zoom:



» Move a câmera:



» Focaliza seleção



» Visualiza todos elementos



3. SELECIONANDO OBJETOS

Com a ferramenta de seleção (**Q**) ativada podemos selecionar ou desselecionar objetos utilizando as os seguintes comandos:

» Adicionar/inverter seleção:



» Desselecionar:



» Caixa de seleção:

Clicar em uma área vazia e arrastar

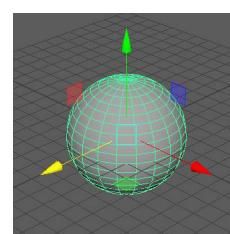
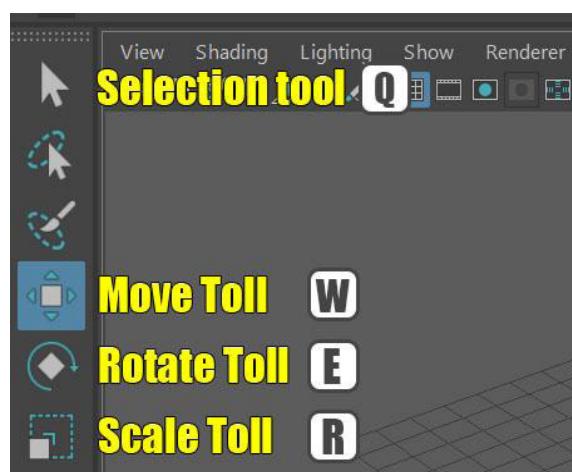


» Avançar/retroceder seleção:

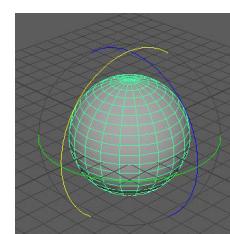


4. MANIPULANDO OBJETOS

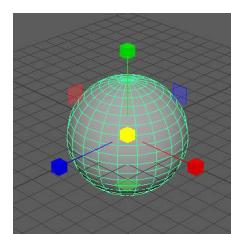
Modelar em 3D consiste, de uma forma simplificada, em posicionar objetos ou seus componente no ambiente tridimensional, boa parte da ferramentas disponíveis servem para automatizar esta tarefa, porém muitas vezes esse processo precisa proporcionar um controle das operações mais básicas, **Translate** (mover), **Rotate** (rotacionar) e **Scale** (escalonar):



Move Tool



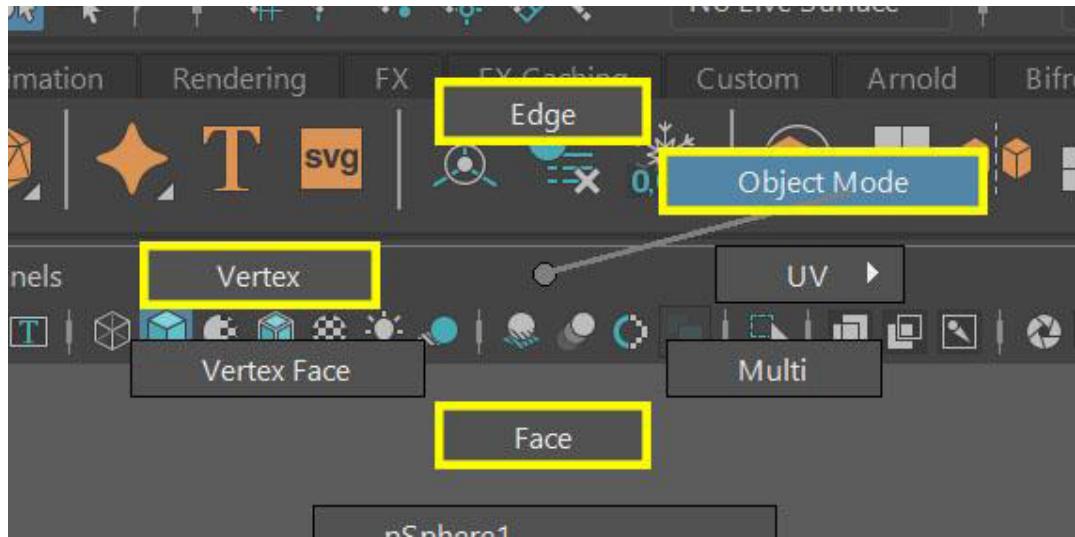
Rotate Tool



Scale Tool

5. MANIPULANDO COMPONENTES

A criação a partir das primitivas é um bom ponto de partida para modelagem, no entanto podemos manipular os elementos que as compõem criando formas mais complexas, para acessar estes elementos pressione o **botão direito do mouse** sobre o objeto que pretende alterar e arraste o cursor para opção desejada:



» **Object mode:**

Manipula o objeto como um todo.

» **Vertex:**

É o componente fundamental, ele cria todos os outros componentes e sua posição determina as formas dos objetos, por ser um ponto de coordenada não rotaciona nem escala podendo ser movido apenas.

» **Edge:**

É a linha composta por dois vertexs.

» **Face:**

É formada por três ou mais vertexs conectados por edges.

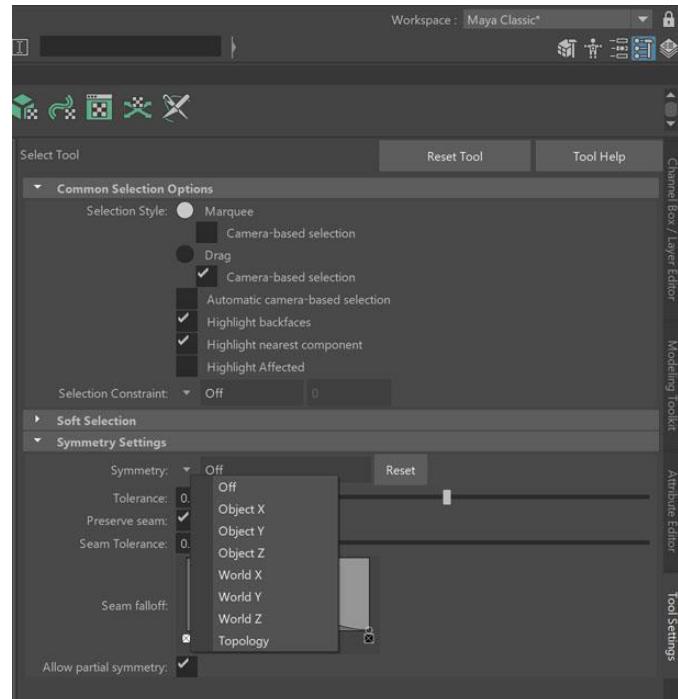
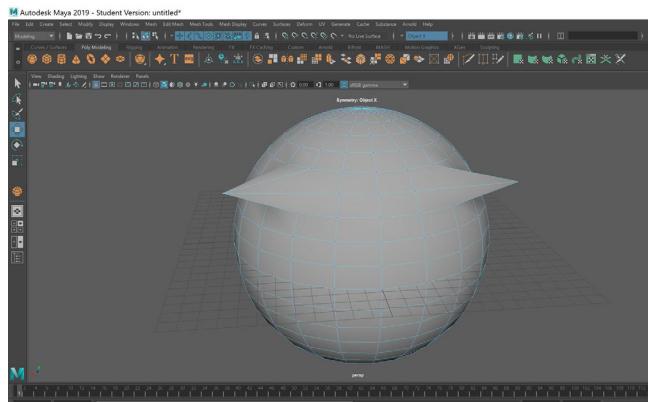
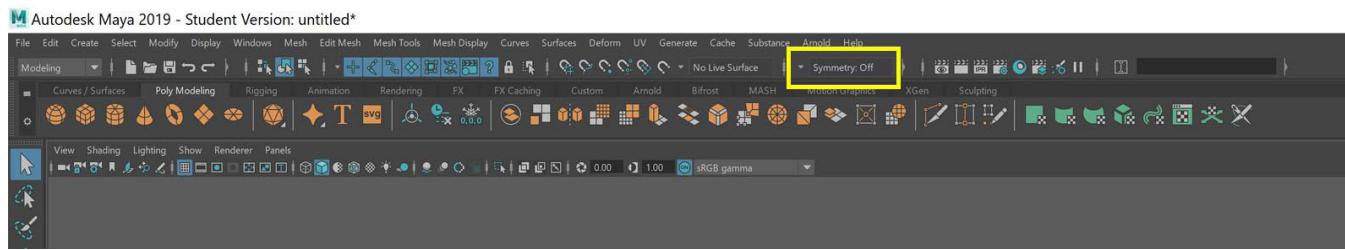
Para acessar as configurações da ferramenta utilizada no momento basta utilizar o **Tool settings**, as opções vão variar de acordo com a ferramenta selecionada.



No caso das ferramentas básicas de manipulação (rotate, move, Scale) podemos destacar duas opções: **Symmetry settings** e **Soft selection**.

» Symmetri settings:

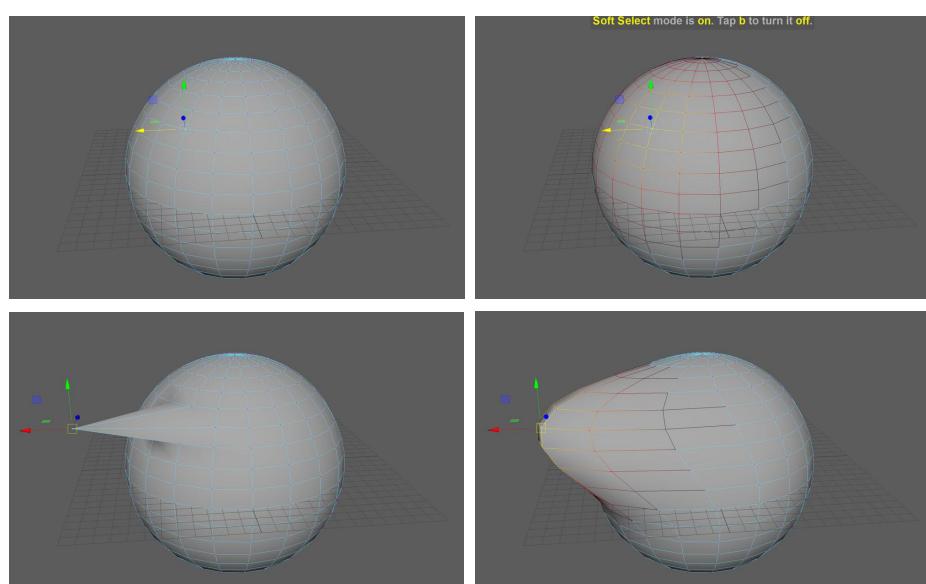
Ferramenta que espelha as transformações feitas nos objetos, agilizando a criação de objetos simétricos, este espelhamento pode ser feito nos três eixos, em relação ao mundo ou de acordo com a topologia do objeto.



- Quando a simetria está ativada os elementos ficam destacados em azul.
- No modo **Topology** deve-se escolher qual o ponto central da transformação.

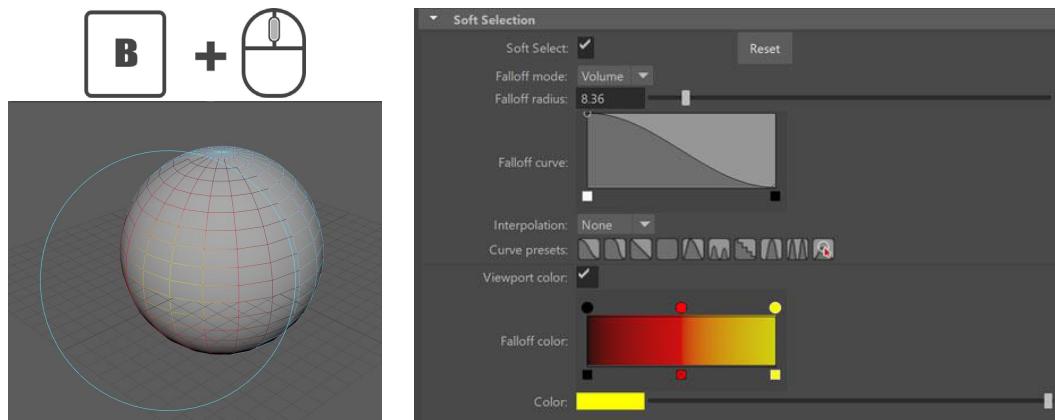
» Soft selection:

Esta ferramenta permite manipular elementos de forma gradual, o que gera uma deformação orgânica, quando o *soft selection* está ativado, os elementos selecionados são manipulados normalmente e os elementos próximos a eles são influenciados de forma gradualmente menor gerando esta deformação suave, são um gradiente de amarelo a vermelho para mostrar visualmente como estes elementos vão ser manipulados.



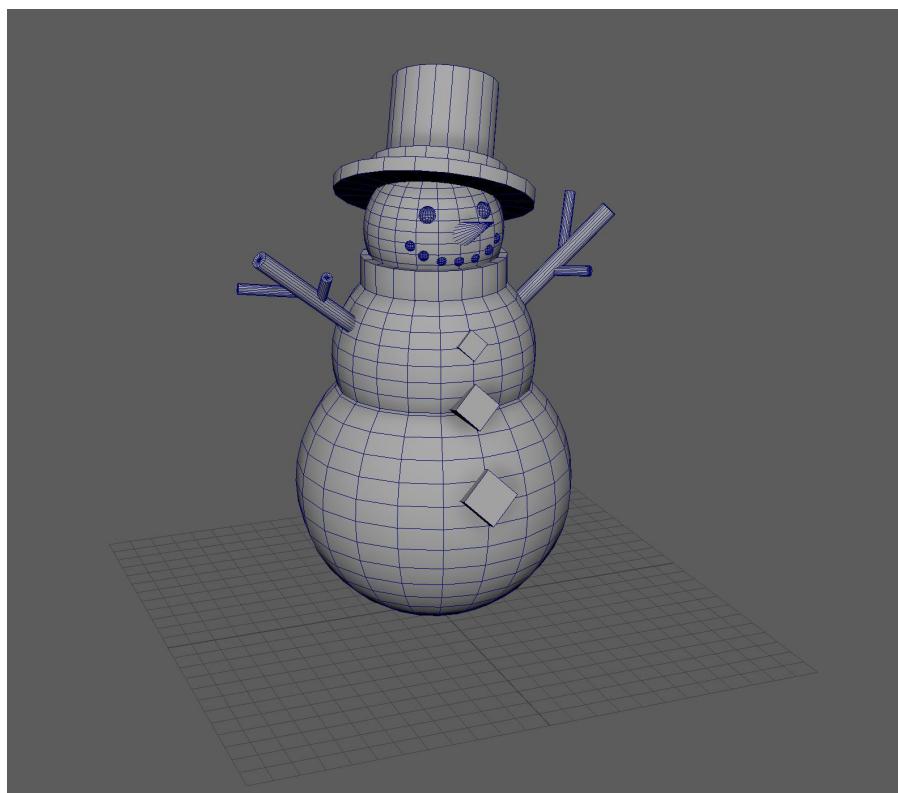
O atalho para ativar o *soft selection* é a tecla **B**.

Para controlar a intensidade de manipulação do *soft selection*, é possível regular a barra **Falloff radius** dentro do *Tool Settings* ou pressionar o **B + botão de rolagem do mouse** e arrastar o mouse lateralmente.



EXERCÍCIO EXTRA - 1

Utilizando apenas primitivas e as ferramentas de manipulação, monte um objeto ou personagem, tente acrescentar o máximo de detalhes possíveis para deixá-lo mais interessante.



Com esse exercício treinamos:

- » manipulação de câmera.
- » ferramentas básicas de manipulação.
- » edição de primitivas.

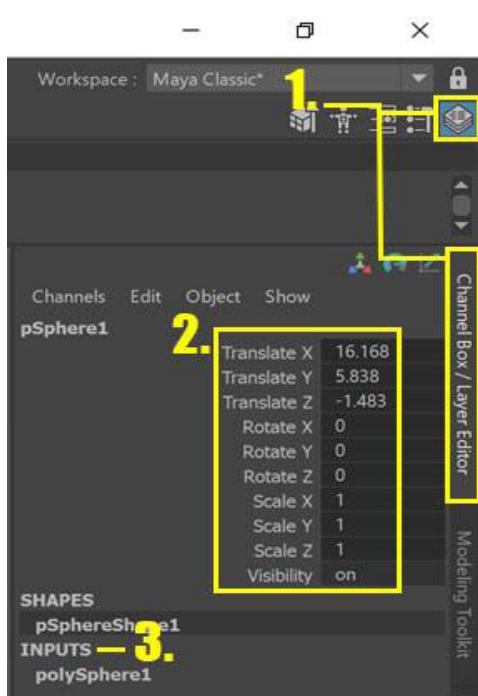
O trabalho pode ser salvo no caminho **File/Save scene (Ctrl+S)**.

POSICIONAMENTO

Nesta unidade vamos falar sobre ferramentas que ajudam a posicionar nossos objetos de forma precisa e a duplicar formas simples. Quando fazemos cenários onde posicionamos e criamos vários elementos semelhantes ou fazemos detalhes que se repetem como botões em uma camisa ou as pernas de uma mesa, o Autodesk Maya nos disponibiliza formas de medir distâncias e agilizar o processo como um todo.

1. CHANNEL BOX

Esta aba mostra informações do objeto selecionado como: posição, escala, entre outros. Podemos fazer transformações digitando nas caixas de cada atributo, o que permite um controle fino sob os mesmos.



01. Locais de acesso ao *channel box*.

02. Atributos do objeto selecionado no momento de acordo com o eixo de transformação.

» **Translate**: posição do objeto em relação ao centro do mundo, este representado pela grid (grade).

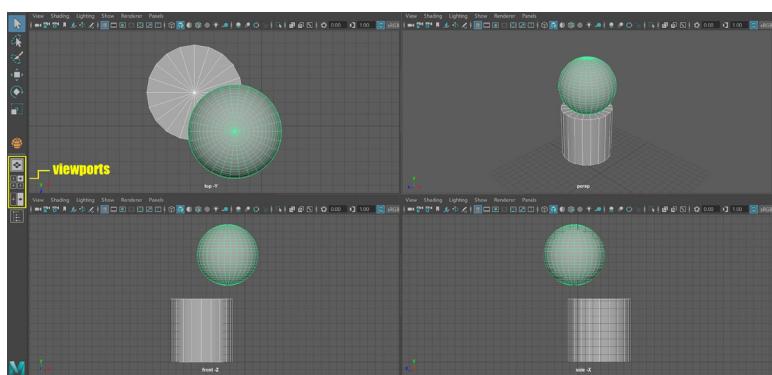
» **Rotate**: Rotação do objeto em graus, podemos usar números com 90 ou 180 para posicionamentos precisos.

» **Scale**: Escala do objeto sendo 1 o tamanho original do objeto, caso queira um objeto com o dobro do tamanho aumente a escala para 2 e assim por diante.

» **Inputs**: Nesta opção fica parte do histórico do objeto, podemos alterar suas configurações iniciais e alguns atributos acrescentados posteriormente.

2. VIEWPORTS

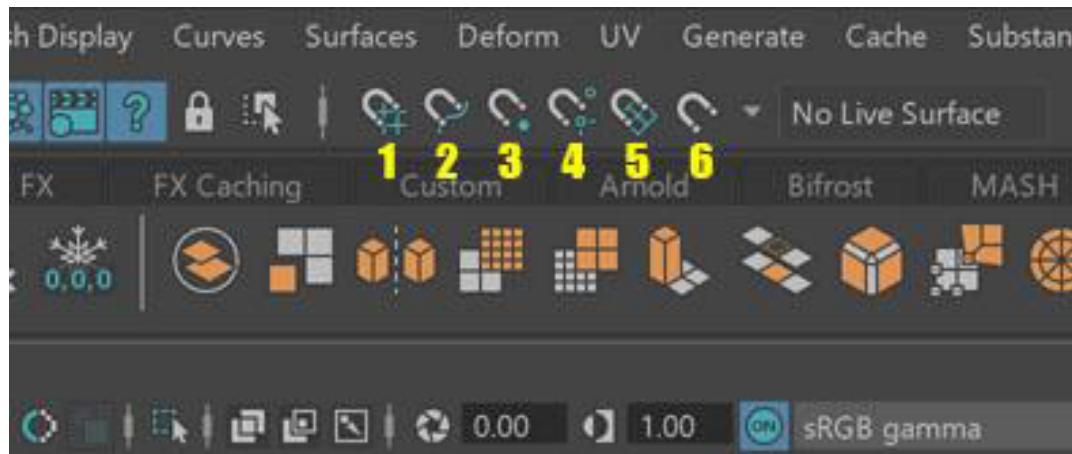
Viewport é como chamamos a visão de nossa área de trabalho tridimensional, basicamente é uma câmera que utilizamos para visualizar nossos objetos, por default a câmera principal é a **perspective**, nela podemos girar e mover a visão em torno do modelo, no entanto ao posicionar os objetos desta forma é preciso ter cuidado para o ângulo de visão não confundir na hora de posicionar os elementos. Para isso existem as vistas ortográficas, **top**, **front** e **side** estas viewports anulam a perspectiva visualizando a cena totalmente voltada para um eixo, o que permite o posicionamento preciso dos objetos.



Para alternar a viewports basta dar um clique na **barra de espaço** com o cursor do mouse na área de trabalho para abrir as quatro vistas e outro clique na barra de espaço com o cursor sobre a tela que quer maximizar, outra forma de alternar ou resetar as viewports é nos botões no centro da barra de atalho da lateral direita do menu.

3. SNAPS

Um snap funciona como um imã que trava as transformações dos objetos ou seus componentes (face, vertex ou edge) em relação a um elemento específico de acordo com o snap selecionado, isto nos permite alinhamentos perfeitos de acordo com nossa necessidade.



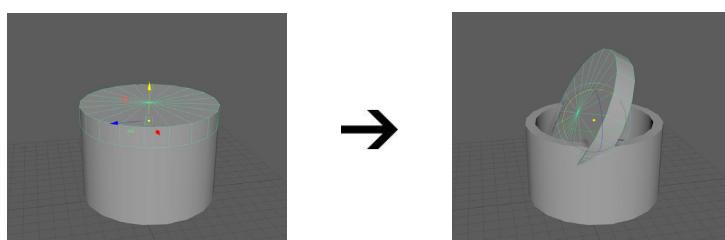
01. **Snap de grid:** Alinha a transformação com a grade, excelente para posicionar elementos em cenários. (ativa mantendo o **X** pressionado).
02. **Snap de linha:** Alinha a transformação com uma linha desenhada previamente. É uma boa opção para fazer um objeto percorrer um caminho.
03. **Snap de ponto:** Alinha a transformação com vértices, conveniente para posicionar elementos em pontos específicos de um objeto. (mantenha **V** pressionado).
04. **Snap ao centro:** Transformações são alinhadas ao centro do objeto. Geralmente usada na criação de esqueletos para animação.
05. **Snap de superfície:** Permite alinhamento com superfícies de objetos.
06. **Snap a objeto:** Transforma um objeto em um centro de transformação, novos objetos podem ser criados a partir dele e várias transformações se mantêm no limite do objeto transformado em snap.

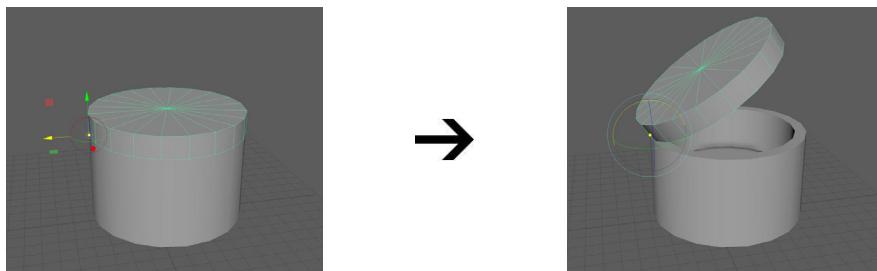
4. PIVOT

O pivô é o ponto a partir de onde as transformações de um objeto acontecem, por padrão ele vem posicionado no centro do objeto, mas podemos editar sua localização para facilitar posicionamentos de objetos ou alterar a forma como o objeto se move e rotaciona.

O caminho para ativar a edição de pivô é **Tool settings > Edit pivot** ou com a tecla **Insert** no teclado.

A posição do pivô é importante para determinar como um objeto se move, ele pode “prender” um objeto em um determinado ponto.





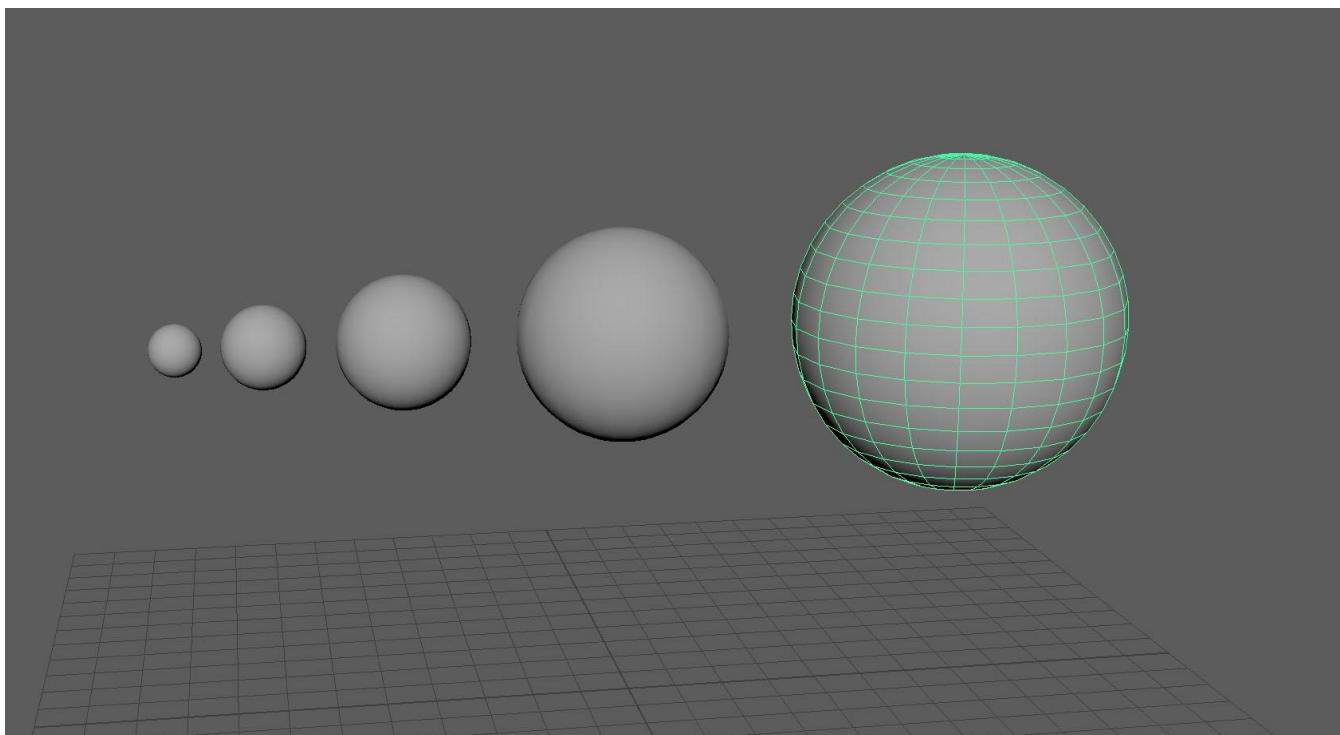
Para retornar o pivô para o centro do objeto o caminho é **Modify > Center pivot**.

DUPPLICATAS

Em muitas situações usamos um mesmo modelo replicado várias vezes em uma mesma cena, coisas como postes em uma avenida ou botões em um mecanismo, para duplicar um objeto use o caminho **Edit > Duplicate (Ctrl+D)**. Quando um objeto é duplicado ele cria uma cópia com o mesmo nome mais com um número adicionado de acordo com a quantidade de cópias. Importante advertir que o tradicional **Ctrl+C > Ctrl+V deve ser evitado** pois o mesmo duplica todos os parâmetros do objeto, o que não costuma ser o desejado.

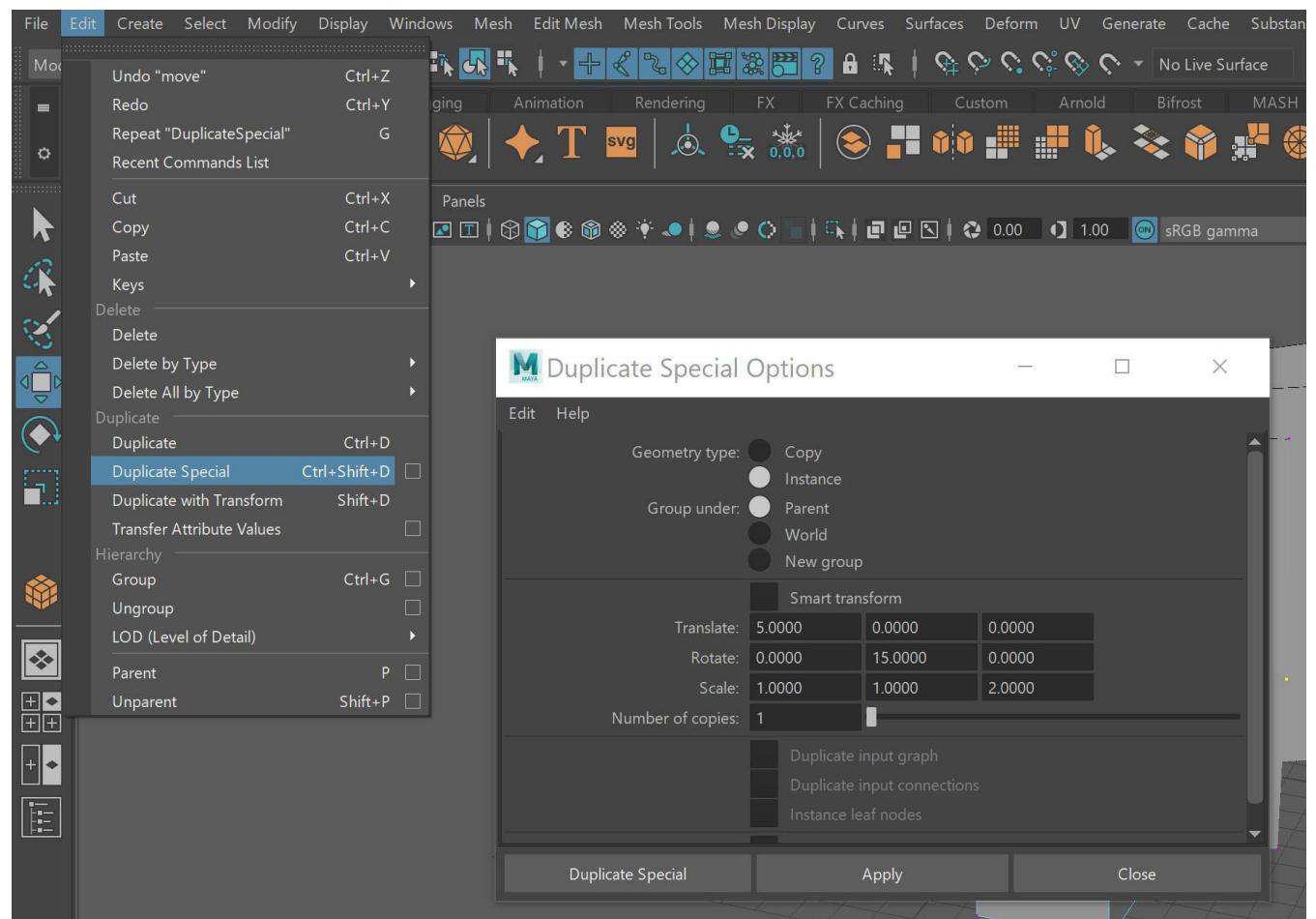
1. DUPLICAR COM TRANSFORMAÇÕES

Existe como fazer duplicatas que adicionam transformações do objeto que está sendo copiado. Para aplicar este tipo de transformação primeiro usa-se a duplicate e em seguida se adiciona as alterações desejadas e usando **duplicate with transformation (Ctrl+Shift+D)**, as próximas duplicatas vão adicionando esta mesma transformação, seja *translate*, *rotate* ou *scale*.



2. DUPLICAR ESPECIAL

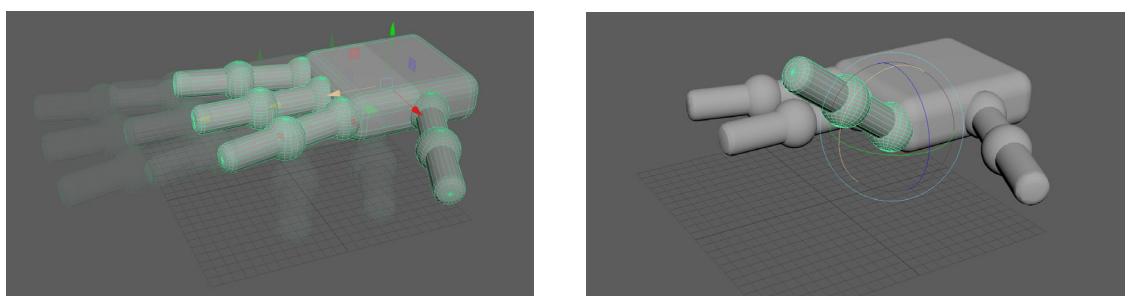
Outra opção para duplicatas é o **duplicate special**. Com esta ferramenta as duplicatas podem ser feitas acrescentando numericamente transformações, além disso é possível criar uma instância, que é uma cópia que acompanha as alterações feitas no objeto original. **Edit > duplicate special (Ctrl+Shift+D)**.



PARENTESCOS

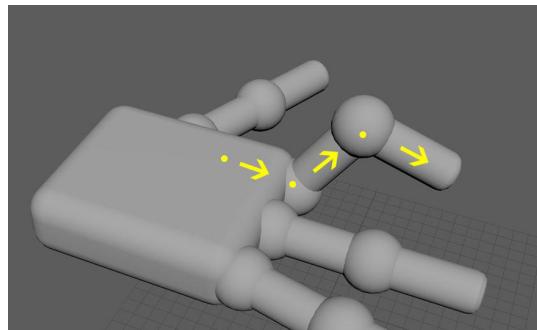
Com parentescos podemos lidar com diversos objetos que funcionam em conjunto, podemos usar de exemplo um piano, ele tem diversas partes como teclas, tampas, pistões, e essas partes juntas fazem o instrumento. O corpo do piano move as as partes presas a ele, mas as teclas por exemplo não movem o piano, porém podem se mover de forma independente, no exemplo, o corpo do piano é o pai, ao se mover todas as peças se movem, quando uma tecla se move o corpo continua parado. Resumindo o pai move o filho, mas os filhos se movem de forma independente não influenciando o pai. Para criar um parentesco, o primeiro objeto selecionado será o filho e o último selecionado será o pai.

o comando para criar o parentesco é **Edit > parent (P)**.



Um pai pode ter vários filhos, quando selecionamos múltiplos objetos e fazemos o parentesco, o último objetos selecionados será o pai de todos os outros objetos.

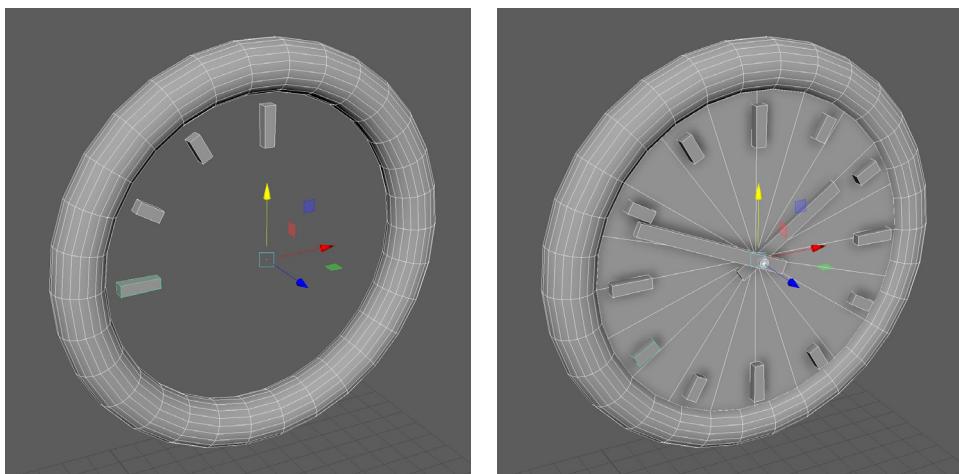
É possível fazer uma cadeia de parentesco, onde temos pais que têm filhos que são pais de outros objetos, criando uma hierarquia, este tipo de estrutura é especialmente interessante para criar movimentos graduais como os vagões de um trem ou penas de uma asa.



Para desfazer parentescos basta selecionar os objetos relacionados e usar *unparent*. **Edit > unparent (Shift+P)**

EXERCÍCIO EXTRA 2

Crie um relógio de ponteiros usando primitivas, utilize os pivôs e snaps para facilitar seu trabalho, duplicate special pode ajudar a posicionar os marcadores das horas e não esqueça de fazer os parentescos para poder mover todo o conjunto e mudar os pivôs dos ponteiros para que eles funcionem corretamente.



Com este exercício aprendemos:

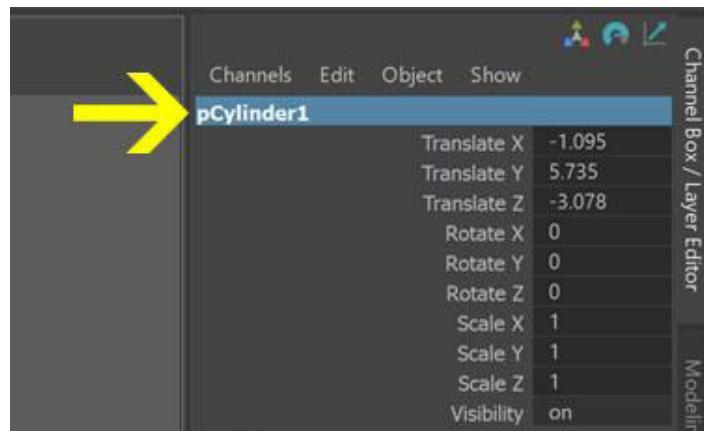
- » Utilizar snaps para posicionar elementos.
- » Alterar pivôs para facilitar posicionamento.
- » Alterar pivôs para modificar transformações dos objetos.
- » Usar parentescos para criar hierarquias de objetos com partes móveis.

ORGANIZAÇÃO

A organização é importante para qualquer produção profissional, além de poupar tempo procurando elementos em uma cena muito complexa, criar padrões ajuda a organizar trabalhos onde diversas pessoas lidam com o mesmo arquivo e evita eventuais *bugs* devido a mau posicionamento ou informações indesejadas.

1. NOMEANDO OBJETOS

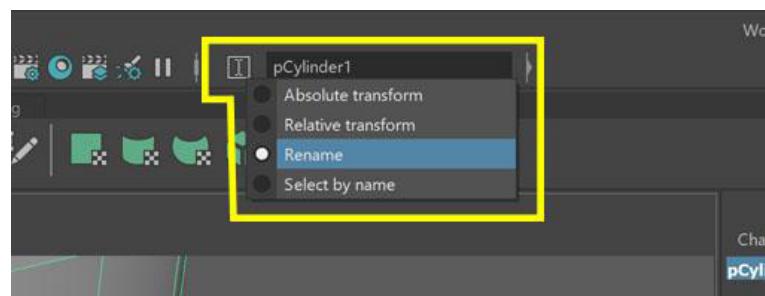
Nomear seus objetos é fundamental, em uma cena complexa, seja na engine em que o jogo for montado ou no próprio Autodesk Maya podem haver centenas de modelos, caso não nomeados podem gerar confusão na hora de lidar com esses. Para alterar um nome de um objeto basta dar dois cliques no seu título no **channel box**.



01. Nomeando múltiplos objetos

Por vezes objetos são repetidos em uma mesma cena, como vários postes em uma calçada ou um conjuntos de árvores, estes objetos geralmente tem nome semelhantes, fazendo necessário renomear uma grande quantidade objetos simultaneamente, existe a ferramenta **rename** para facilitar este processo.

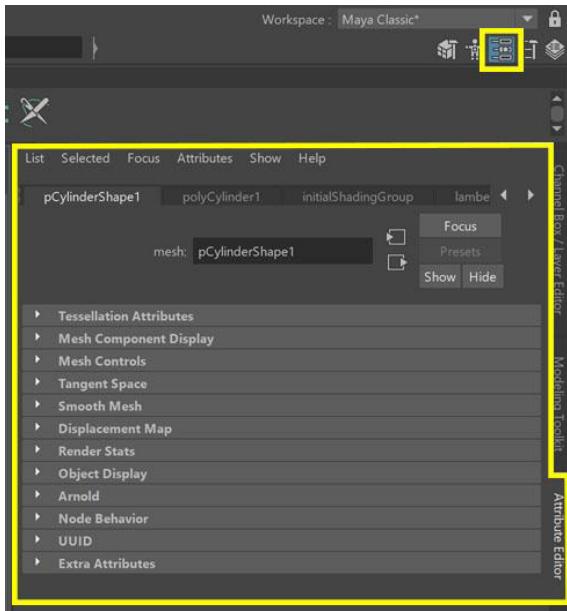
Esta ferramenta troca o nome enumerando os objetos selecionados, respeitando a ordem em que esta seleção foi feita.



2. HISTÓRICO

Muitas das transformações que são feitas nos objetos são armazenadas em seu histórico, sendo possível desfazer (tecla **Z**) ou refazer (**shift+Z**) estas transformações. Cada objeto tem seu próprio histórico que pode ser acessado no **Attribute Editor** o que nos permite fazer alterações um uma determinada transformação feita anteriormente sem ter que desfazer o processo até aquele ponto específico.

Apesar de ser muito útil, o histórico precisa ser deletado em duas ocasiões.



» Quando uma cena tem muitos objetos com históricos carregados o computador pode não conseguir manter a performance, limpar históricos pode evitar essa situação.

» Quando fazemos jogos normalmente o montamos em um outro software, ao mandarmos modelos 3d para estes softwares o histórico é informação indesejada apenas pesando o arquivo e podendo ocasionar bugs.

O caminho para deletar o histórico do objeto selecionado:

Edit > Delete by type > history (Alt+shift+D).

Para deletar o histórico de todos os objetos da cena:

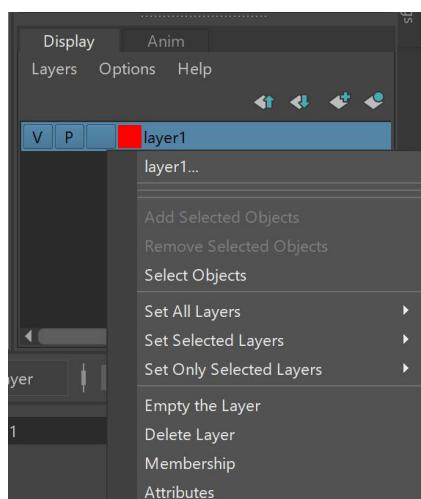
Edit > Delete all by type > History.

3. FREEZE TRANSFORMATIONS

Como vimos anteriormente o channel box mostra as alterações de posição rotação e escala, além de parte do histórico de um objeto, estas transformações podem atrapalhar novas modificações no objetos ou dificultar a implementação do objeto em um jogo, para resolver este problema existe a opção *Freeze transformation*, que zera as transformações do objeto tornando seu estado atual o padrão para aquele modelo. **Modify > Freeze transformations**

4. LAYERS

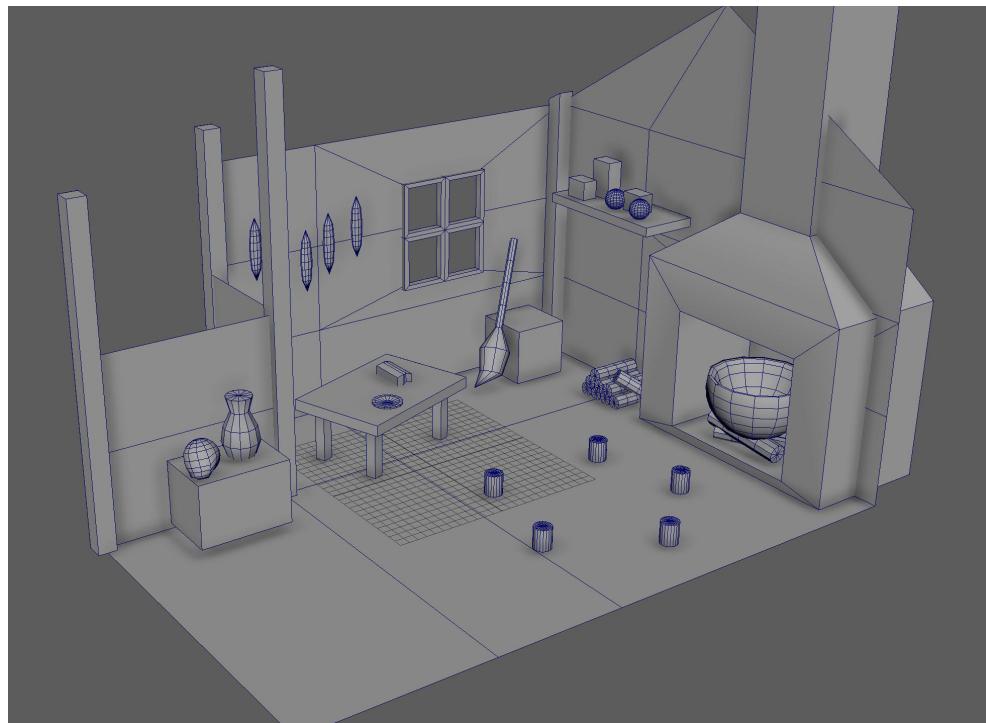
Outra forma de manter a organização na sua cena é criar *layers*, isto cria camadas de seleção e visualização que permite agilizar estes processos, especialmente quando se trabalha com múltiplos objetos que podem ser colocados em uma layer os isolando dos demais. O caminho para criação de uma layer é **channel box/layer editor/layer/create empty layer** ainda é possível no mesmo caminho criar uma **layer from selected** que já coloca os objetos selecionados no momento dentro da nova layer.



- » Clicando com o botão direito do mouse em cima de uma layer existente é possível adicionar ou remover objetos selecionados.
- » Dando dois cliques na layer selecionada pode-se atribuir uma cor a camada e alterar seu nome.
- » existem 3 quadrados no início de cada layer, no primeiro coloca ou tira a visualização dos elementos da camada **V**, o segundo faz quem que os elementos sejam ou não visualizados durante uma animação **P**, na terceira caixa muda o modo de visualização e seleção dos objetos, **T** os deixa transparentes e não selecionáveis e **R** somente não selecionáveis.

EXERCÍCIO EXTRA 3

Crie um pequeno cenário com objetos simples, use layers para ajudar na organização e visualização da cena, especialmente nas paredes e chão, ter a possibilidade de remover a visualização de paredes específicas sem precisar a remover do local e poder selecionar os objetos sem ficar clicando no chão ou paredes acidentalmente, lembre-se de dar nome aos objetos, setar suas posições com freeze transformations, e limpar o histórico ao final do processo, crie o hábito de fazer isto em todos seus projetos.



Com este exercício aprendemos:

- » Facilitar manipulação e visualização de múltiplos objetos.
- » Organização geral de uma cena.

Você pode importar objetos feitos em exercícios anteriores para compor seu cenário, basta ir no caminho **file/import** e você importa uma cena para dentro da outra.

FERRAMENTAS DE MODELAGEM

Neste ponto começaremos a fazer transformações mais profundas em nossos objetos, veremos ferramentas que modificam e criam malha de forma mais complexa. Para fins didáticos vamos dividir as ferramentas pelos menus onde se encontram e citar as mais importantes.

1. MESH

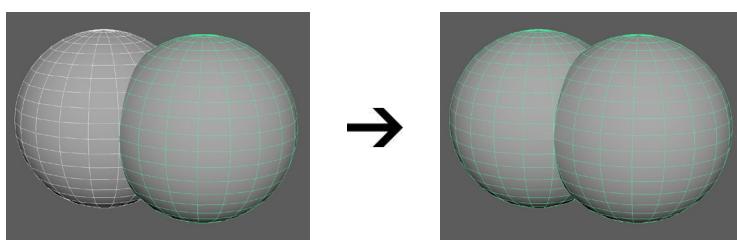
As ferramentas do menu edit mesh costumam trabalhar com um determinado modelo por inteiro ao invés de modificar pontos específicos e fazer correções em uma malha.

01. **Booleans**

As ferramentas de booleanas trabalham de forma semelhante ao que estudamos em conjuntos na matemática (união, intersecção e diferença), podemos unir dois objetos ou subtrair parte de um objeto de um outro.

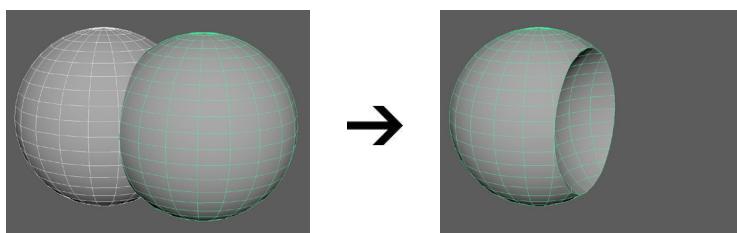
» **Union:**

A Ferramenta *union*, junta dois ou mais objetos em um único modelo, deletando as faces internas desta junção. É indicado quando economiza na quantidade de polígonos ou quando a malha precisa existir no ponto de junção entre as formas, no entanto em alguns casos pode acabar aumentando a quantidade de polígonos ou gerando um fluxo ruim de malha.



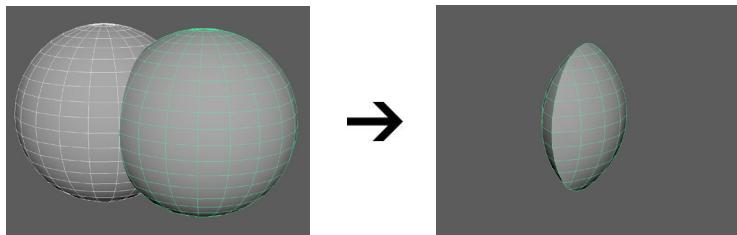
» **Difference:**

A operação *difference* subtrai de um modelo as partes que coincidem com outro objeto selecionado. É um jeito rápido de criar buracos e formas mais complexas, mas como no union deve se ter cuidado com a topologia do objeto final.



» **Intersection:**

A *intersection* cria um modelo a partir das partes de coincidem entre os objetos selecionados. Assim como as ferramentas anteriores agiliza a construção de formas complexas, mas é necessário ficar atento com o fluxo de malha.



02. **Combine e Separate**

O comando *Combine* serve para transformar múltiplos objetos em um único modelo, mas ao contrário do *union* não altera o formato original de nenhuma das formas; já o *Separate* desfaz esta operação voltando os objetos ao estado original.

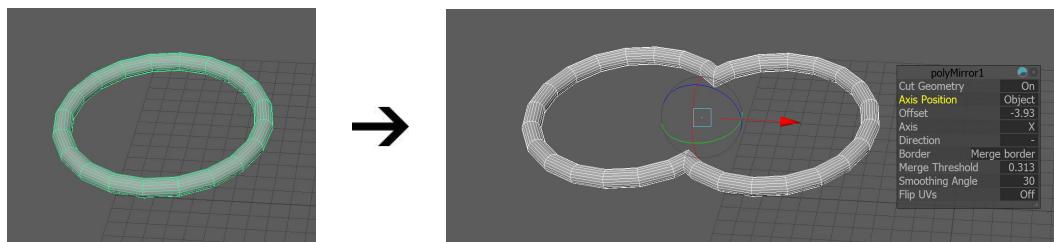
03. Smooth e Reduce

A ferramenta *smooth* é utilizada para aumentar a densidade de polígonos de um modelo, tentando suavizar o contraste das formas. Um fluxo possível de trabalho é fazer os modelos com poucos polígonos e depois usar *smooth* para finalizar o modelo, porém em jogos esta é uma técnica pouco indicada, como a *smooth* aumenta a densidade da malha de forma uniforme, sua utilização pode colocar polígonos em áreas onde eles não são necessários.

O comando *reduce* faz o oposto do *smooth*, reduzindo em porcentagem a quantidade de malha de um modelo, tentando manter o formato da melhor forma possível de acordo com a porcentagem definida.

04. Mirror

A Ferramenta *Mirror* cria uma cópia espelhada de um objeto, esta cópia pode cortar parte da malha original, sendo este comando muito utilizado para criar um objeto simétrico, onde se modela apenas metade do objeto depois se faz o espelhamento. O atributo **offset** escolhe o ponto a partir de onde o espelhamento é feito, o **axis** escolhe o eixo que é afetado pelo *mirror* (X,Y ou Z) e é importante se atentar ao **direction** que é o sentido para onde será feito este espelhamento.

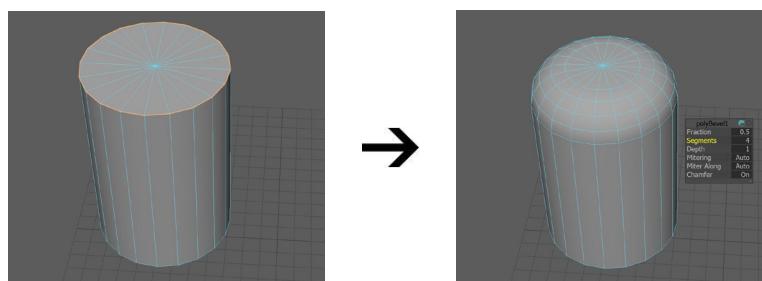


2. EDIT MESH

As ferramentas do menu *edit mesh* tendem a fazer transformações pontuais, voltadas aos componentes (face, vertex ou edge), criando ou movendo estes. Sendo os itens deste menu muito importantes em modelagens mais complexas, que exigem detalhes mais finos e menos automatizados, se comparados ao menu *mesh*.

01. Bevel (Ctrl+B)

Este comando é utilizado para aumentar densidade de malha e suavizar as formas, mas ao contrário do *smooth* ele trabalha melhor em pontos específicos de malha ao invés da malha inteira, o que é muito vantajoso na produção de jogos já que a malha aumenta de forma localizada. O atributo **fraction** determina a distância total da suavização da malha em relação ao formato original, já o **segments** controla o número de divisões feitos pelo comando e o **depth** controla como a suavização se comporta.

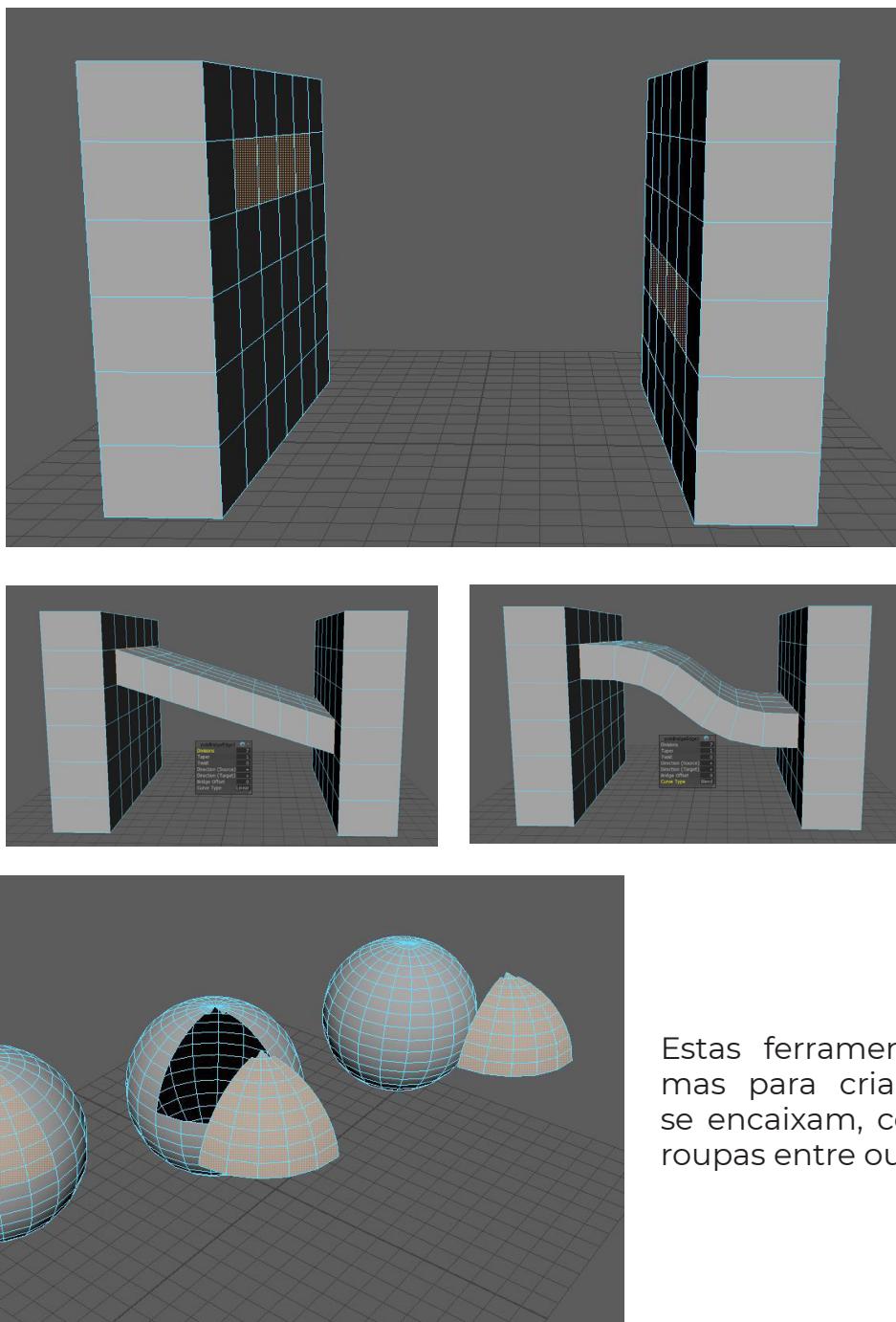


02. Bridge

O comando Bridge conecta faces ou edges de um mesmo objeto, como o nome sugere, criando pontes. A forma como essa ferramenta se comporta varia muito de acordo com o formato e distância entre os componentes. O atributo **Divisions** determina quantas divisões na malha são adicionadas na conexão, **Taper** controla a espessura no centro da conexão, o atributo **twist** torce a malha que está sendo criada, **Bridge offset** regula a deformação da ponte como um todo e **Curve type** define o tipo de suavização utilizada na conexão.

03. Detach, Extract e Duplicate

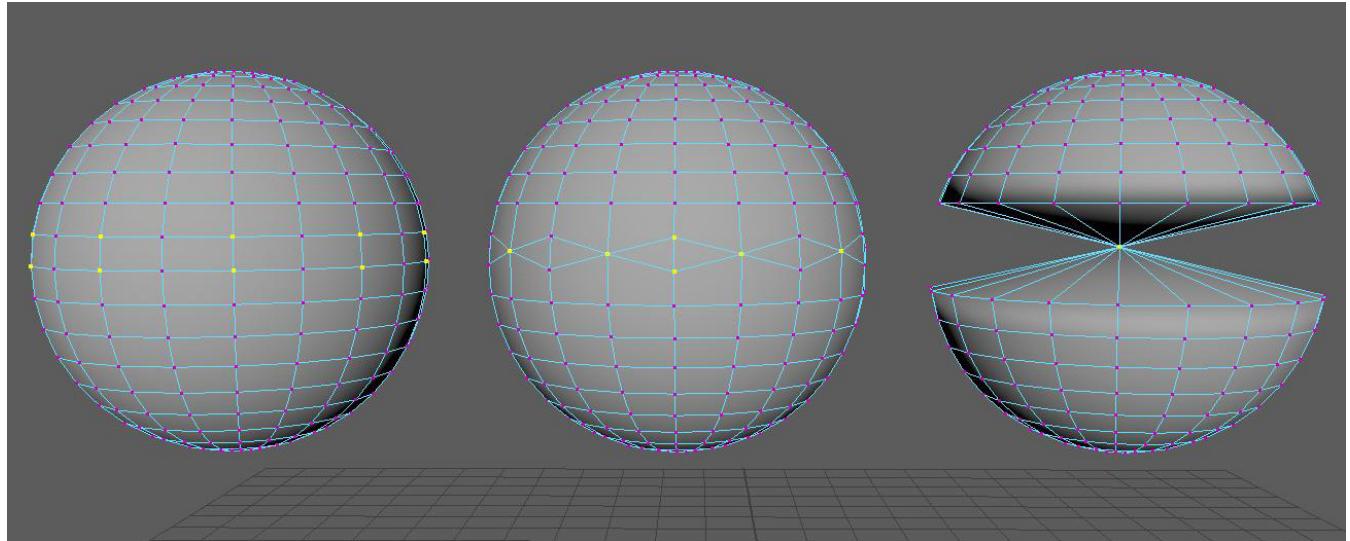
Essas três ferramentas fazem coisas parecidas porém cada uma tem sua aplicação. A **Detach** separa as faces selecionadas de um objeto, no entanto estas faces continuam pertencendo ao objeto, já a **Extract** separa as faces selecionadas criando um novo objeto com estas faces e o **Duplicate** cria um novo objeto a partir das faces selecionadas sem subtrair as faces do objeto original.



Estas ferramentas são ótimas para criar peças que se encaixam, como tampas, roupas entre outros.

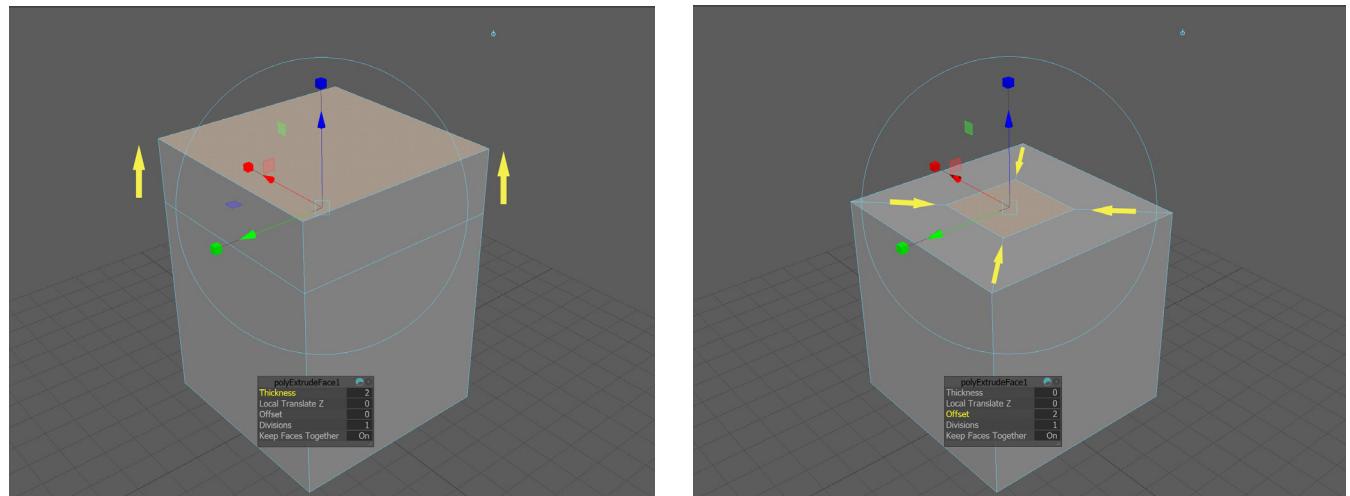
04. Merge e Merge to center

Estas ferramentas servem para “colar” elementos como vertices ou edges, fazendo que estes virem um único componente, a diferença entre as duas ferramentas é que a **merge** tem uma distância máxima em que os elementos se unem o que permite juntar vários elementos ao mesmo tempo sem que todos se tornem um único ponto e a **Merge to center** junta os elementos independente da distância entre eles.



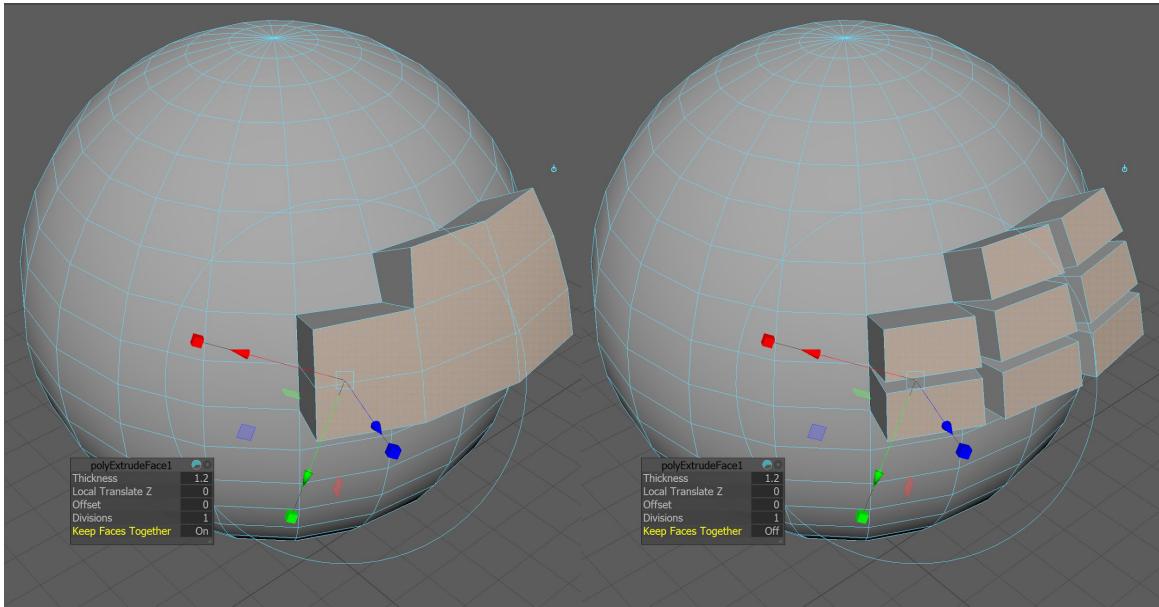
05. Extrude

Esta é uma das ferramentas mais ágeis e versáteis, capaz de criar formas complexas de forma rápida e simples. Basicamente a ferramenta pega as faces ou edges selecionadas e as deslocam da posição inicial criando uma conexão com suas faces vizinhas, esta operação é simples porém é possível conseguir formas bastante complexas. Seus atributos são: **Thickness** controla a distância que o componente se distancia do ponto original, **Offset** que é o quando os elementos selecionados expandem ou retraem do ponto de início, **Divisions** acrescentam divisões na conexão criada e **keep faces together** que transforma as faces selecionadas separadamente ou não.



thickness

offset



keep faces together (on/off)

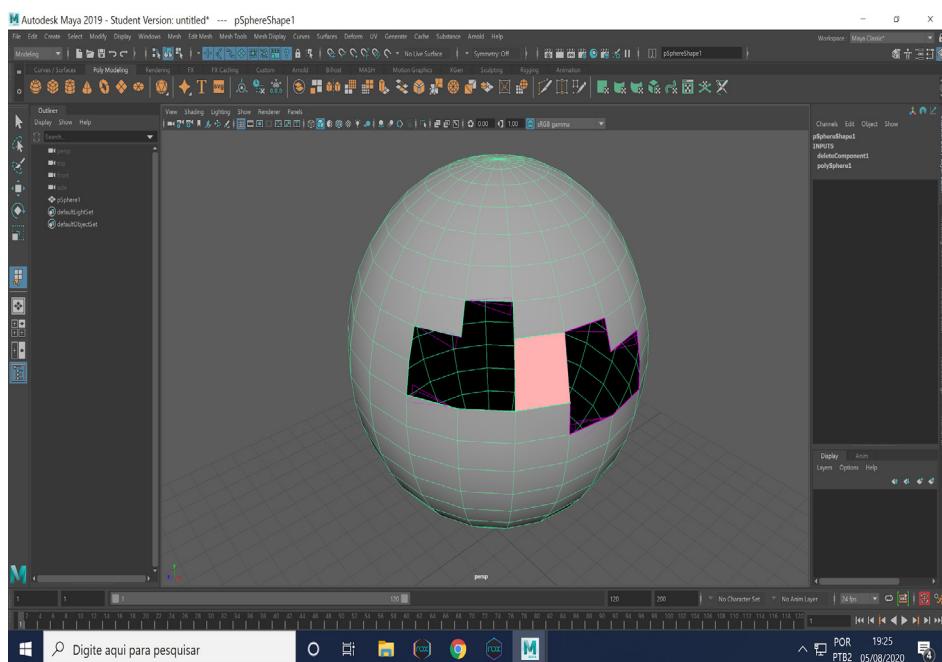
Fazendo vários comandos seguidos, é possível ir dando forma, modelando, controlando *thickness* e *offset* criando coisas como garrafas, caixas e praticamente qualquer objeto de peça única.

3. MESH TOOLS

No menu *Mesh tools* temos na maioria das vezes, ferramentas que são ativadas e podem ser utilizadas até que concluam seu propósito ou serem desativadas, junto as opções dos menus *mesh* e *edit mesh* fechamos as ferramentas básicas mais comuns na modelagem de objetos.

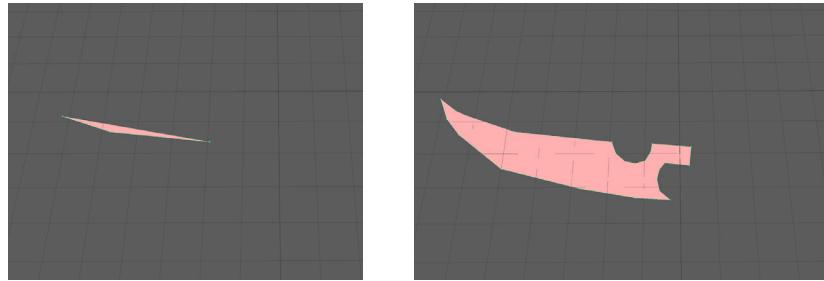
01. Append to polygon

A ferramenta serve para fechar buracos na malha a partir das edges, utiliza um processo manual, mas permite maior controle de direção e área. Primeiramente escolha uma das edges destacadas (as que delimitam as aberturas na malha), as edges disponível para completar a ação são sinalizadas em rosa, clicando vai se criando novas faces, a tecla *delete* desfaz a última ação e a tecla *enter* finaliza a operação.



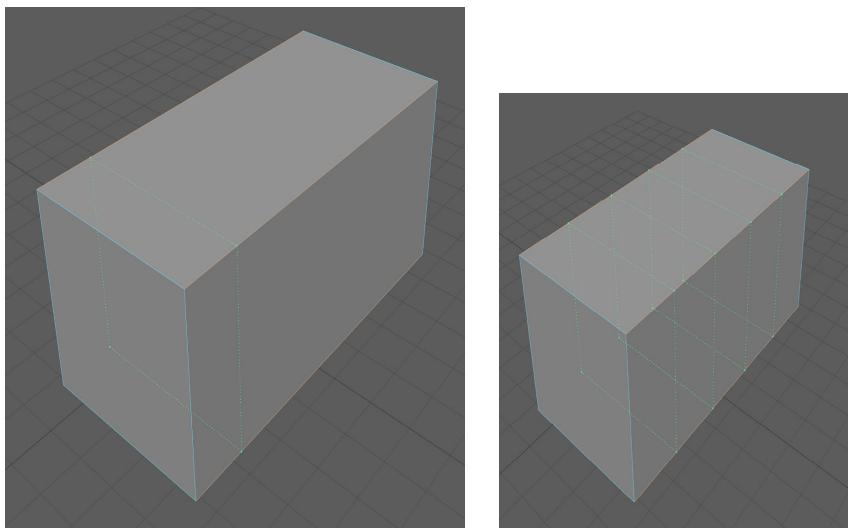
02. Create Polygon

Esta ferramenta é utilizada para criar uma face com formato criado pelo posicionamento de vértices, é eficiente para criar formas planas porém complexas, como lâminas de armas elaboradas por exemplo. como na maioria das ferramentas do menu *mesh tools* a tecla **delete** desfaz os últimos movimentos e o **enter** conclui a utilização.



03. Insert edge loop

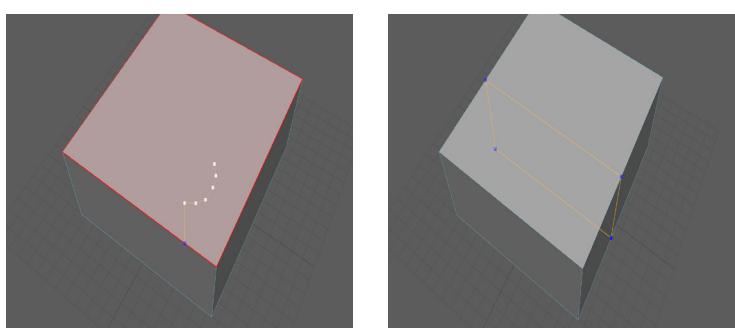
Um erro muito comum aos iniciantes em modelagem é aumentar a densidade da malha para começar a modelar, a quantidade elevada de componentes costuma atrapalhar especialmente os inexperientes, o recomendado é inserir componentes apenas nos locais necessários, a ferramenta *Insert edge loop* é uma das recomendadas nestes caso, clicando em uma edge ela insere um looping no eixo oposto, é possível escolher um ponto específico da malha para esta inserção.



Vale ressaltar o opção *multiple edge loopings*, acessada no *tools settings* desta ferramenta, esta permite inserir vários looping simultâneos com a mesma distância entre si.

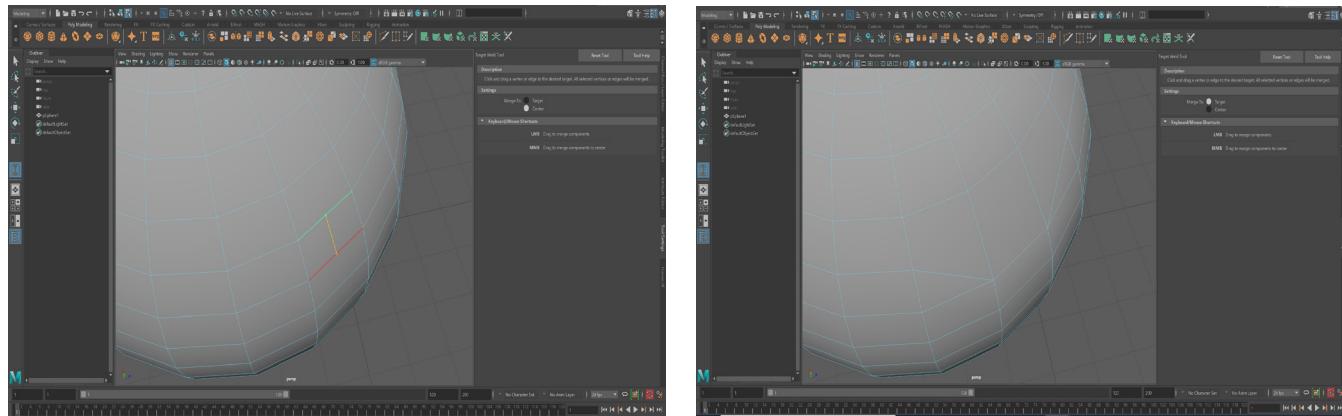
04. Multi-cut

Esta ferramenta cria *vertices* e *edges* livremente em uma malha já existente, seus usos mais comuns são criar um desenho específico na malha ou acrescentar loopings semelhante a ferramenta *insert loop tool*. Com a tecla **shift** é possível acrescentar um *vertex* no centro de uma *edge*, **Delete** desfaz os pontos criados, **Esc** cancela todos os pontos, **Ctrl** adiciona um looping na malha e **Enter** finaliza a criação.



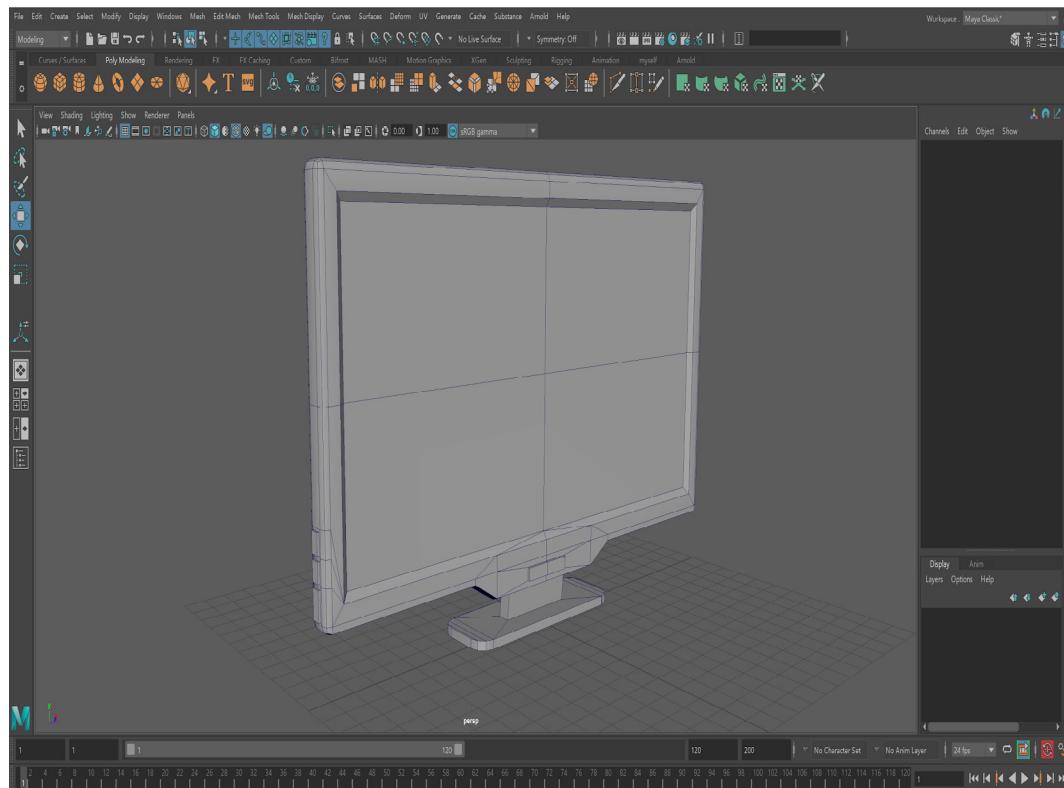
05. Target weld

É uma ferramenta utilizada para fundir *vertices* e *edges*, funciona de forma semelhante ao comando *merge*, por ser uma ferramenta, é recomendada quando necessários múltiplos usos consecutivos, além disso é possível configurar no *tool settings* para que a ferramenta junte os elementos de forma centralizada **to center** ou funde o primeiro selecionado para o segundo **to target**.



EXERCÍCIO EXTRA 4

Modele um eletrodoméstico, procure utilizar as ferramentas utilizadas neste capítulo, especialmente **extrude**, **multi-cut** e **bevel**. Evite deixar faces com mais de quatro lados.



Com este exercício aprendemos:

- » Construir malhas complexas a partir de primitivas.
- » Agilizar o processo de modelagem conhecendo as ferramentas disponíveis.

MATERIAIS E MAPA UV

A superfície que recobre os wireframes (*shade*) é definida por várias características presentes nos materiais entre outros elementos. Além de ditar as cores dos materiais trazem uma série de elementos que ajuda a dar complexidade às superfícies auxiliando a definir de qual material o objeto é feito e seu estado, atributos como reflexividade, transparência entre outros. Portanto compreender como funcionam os materiais aprimora muito as capacidades de um modelador.

Quando trabalhamos para produção de games, normalmente detalhamos os materiais no software em que montamos os jogos, no entanto é interessante ter uma noção básica de como os materiais funcionam, pois muitos dos atributos funcionam de forma semelhante em diversos softwares diferentes.

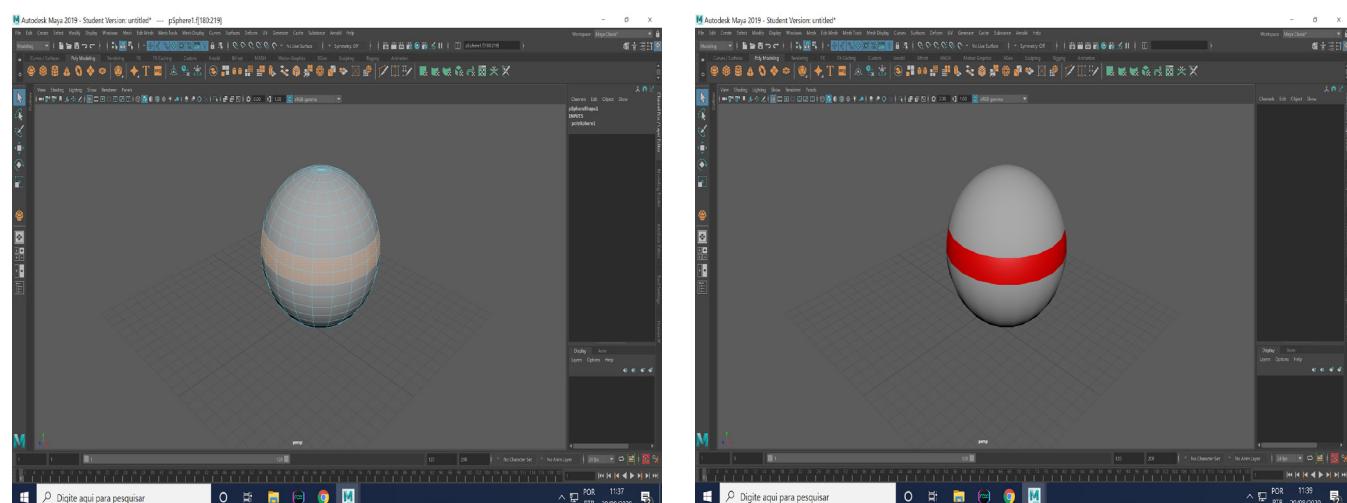
Podemos acessar os materiais atuais do objeto na última aba do menu **Attribute editor**. Por padrão os objetos vêm com o material *Lambert1* anexado a ele, evite fazer alterações neste material pois todos os objetos criados trarão estas mesmas alterações.

01. Anexando novos materiais

Existem duas principais formas de criar e anexar novos materiais aos objetos, a primeira forma é acessando o menu no caminho **windows/rendering editors/hyper-shade**, este menu permite que veja de forma detalhada os materiais e todos que foram criados, possibilitando gerenciar estes materiais, apertando o botão de rolagem sobre um material e arrastando para cima do modelo, é feita a conexão entre os mesmos.

A segunda maneira de anexar um novo material é mantendo pressionado o botão direito do mouse sobre o objeto em que quer fazer a alteração e avançar o cursor até a opção **assign favorite material**, onde existe uma seleção dos materiais mais usados, sendo mais recomendado utilizar um *lambert* ou o *blinn*, em **assign new material** o menu *hypershade* é aberto e em **assing existing material** é possível anexar um material criado anteriormente, o que é muito útil.

É possível colocar um material apenas em faces específicas, ou seja um único objeto pode ter vários materiais diferentes.

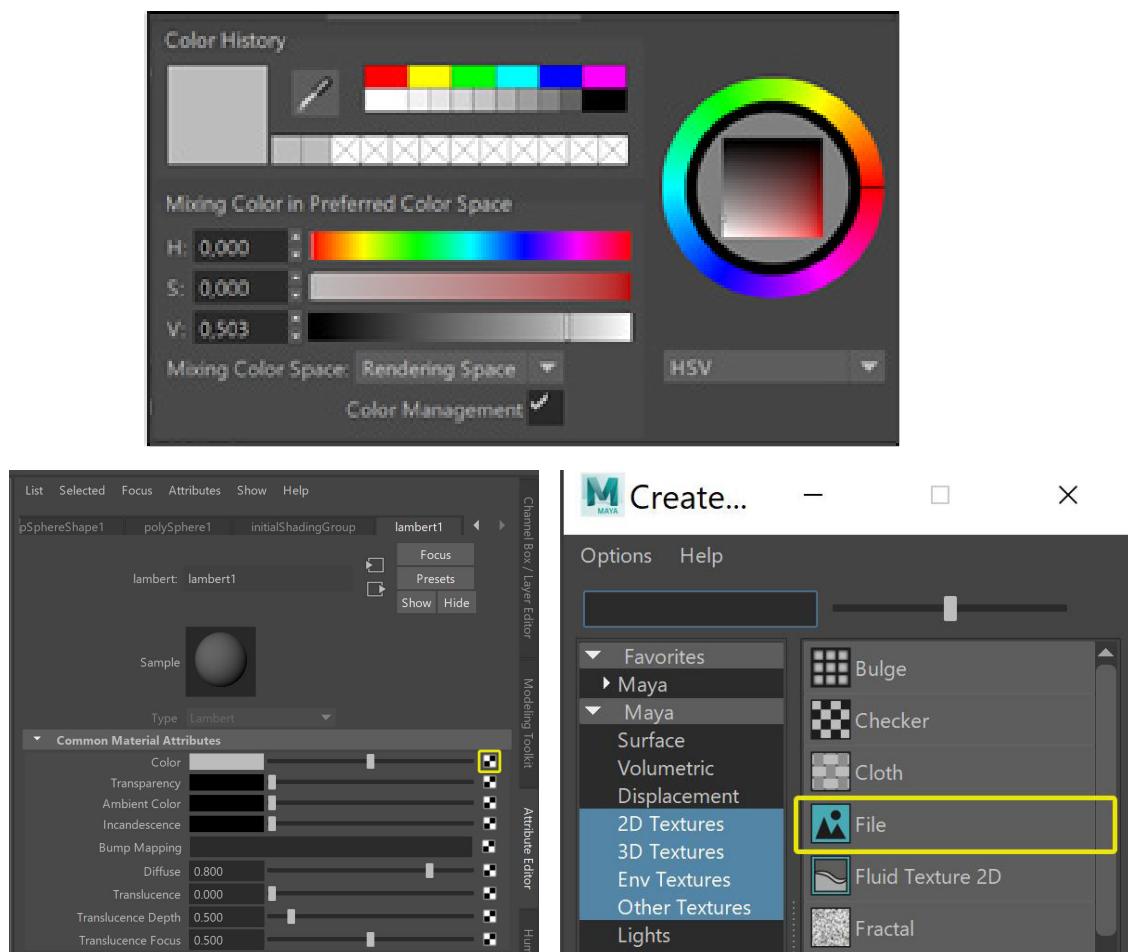


02. Características dos materiais

Acessando as últimas abas do *attribute editor*, visualizamos os materiais anexados ao objeto, nessas abas é possível nomear estes materiais e configurar os seus diferentes atributos, clicando do quadriculado acessamos várias configurações pré definidas. caso precise voltar o atributo ao padrão basta clicar com botão direito do mouse e selecionar a opção **break connections**.

a. Color

Controla as cores visualizadas no objeto, geralmente é a base do material, sendo um atributo importante, sua funcionalidade mais básica é alterar a cor do material como um todo clicando no display da cor atual é aberto o **color history**, onde se pode alterar a cor ou selecionar uma cor em outra janela com o **color picker**.



O uso mais comum deste atributo é anexar um arquivo de imagem ao material, utilizando a opção **file**. a forma como esta imagem é aplicada ao objeto é definida por seu mapa **UV**.

b. Transparency

Atributo que controla a transparência do material, também denominada em muitos softwares como *alpha*, ou opacidade. Funciona em uma escala de cor que varia de preto, cor completamente sólida, ao branco, completamente transparente, uma escala de cinza regula o meio termo entre estes dois pontos. É possível controlar esta transparência através de uma imagem em tons de cinza, fazendo com que pontos específicos do material fiquem transparentes e outros não de acordo com tonalidade da imagem.

c. Ambient color

Simula a luminosidade refletida pelo ambiente, não reproduz o efeito dinamicamente e sim aplica uma cor geral a este efeito, exemplificando, um personagem está em uma floresta e se coloca um pouco de verde em todo modelo utilizando este atributo.

É mais comum trabalhar este efeito com iluminação e efeitos de pós produção na engine onde o jogo esteja sendo montado.

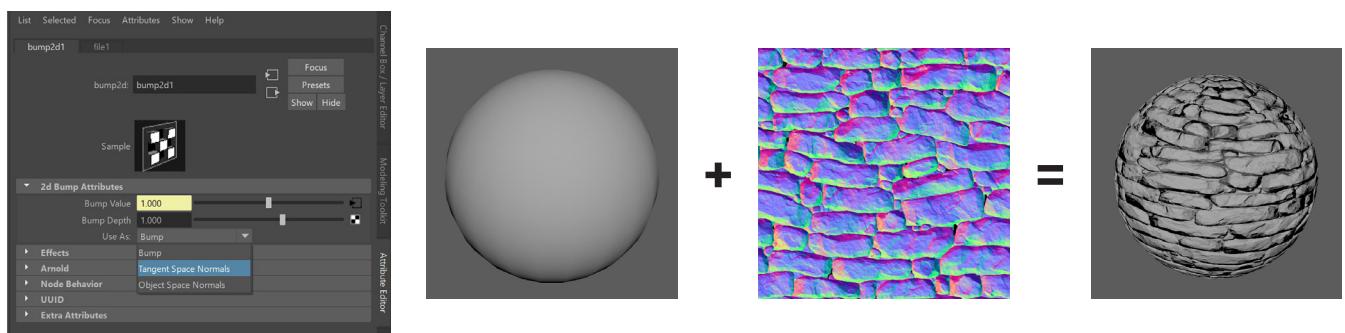
d. Incandescence

Também denominado como emissivo, simula luminosidade em um objeto, não é uma luz propriamente dita e sim um efeito no próprio material. Funciona de forma semelhante a transparência onde quanto mais escuro menos luminoso e quanto mais claro mais brilhante, ou seja a cor preta não gera luz e a branca gera luz plena, no entanto é possível trabalhar cores diversas regulando a luminosidade com tons claros ou escuros.

Existe a possibilidade de usar uma imagem para colocar o efeito em pontos específicos e cores diferentes em um mesmo material.

e. Bump mapping

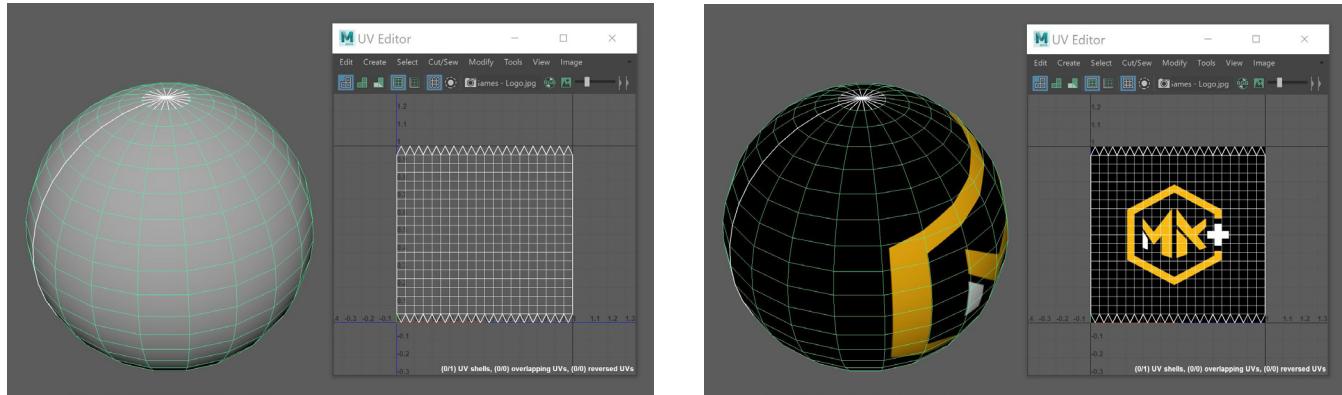
Este atributo permite inserir mapas que simulam volume nas superfícies dos objetos controlando como a luz os influencia. Os dois principais tipos de mapas utilizados neste atributo são o **bump map** que utiliza mapas em escala de cinza para trabalhar elevações ou aprofundamentos no material e o **normal map** sendo este mais complexo já que reproduz o efeito de volume em todos os eixos (X, Y e Z) utilizando um mapa de três cores específicas. Para visualizar o normal map corretamente é necessário usar o bump map como **Tangent Space Normals**.



03. Mapa UV

O mapa UV é uma representação 2D de um objeto 3D, fazendo uma comparação simples o modelo seria um *papercraft* montado e o mapa UV a chapa de papel antes dos recortes e dobraduras. Os polígonos criados a partir de primitivas têm UV's predefinidas por seu tipo, este mapa pode ser visualizado no menu **UV/UV Editor**.

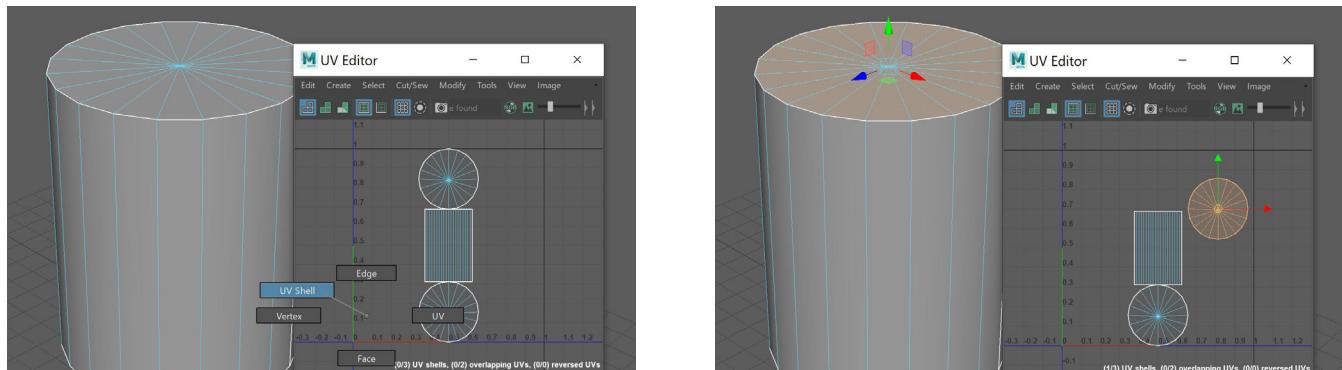
As imagens aplicadas a UV são quadradas, mesma altura e largura, caso coloque uma imagem em outra proporção a mesma vai ser deformada para se encaixar nesse padrão, a posição da imagem onde as linhas brancas que representam a UV definem qual parte da imagem será visualizada nas faces correspondentes.



Na maioria dos casos o mapa padrão que vem com a primitiva, não representa as proporções exatas dos objetos a que correspondem, além disso quando modelamos, criando e deslocando partes da malha, este mapa precisa ser recriado levando em consideração a forma final do objeto, um processo que normalmente chamamos de abertura de UV.

a. UV editor

O menu UV editor permite a visualização e manipulação das UV's, que são pontos que simbolizam a versão 2D das faces. Sua navegação funciona de forma semelhante a uma vista ortográfica, mostrando o mapa UV do objeto selecionado no momento, é possível selecionar partes específicas como faces, vértices ou edges do modelo a partir deste editor, no entanto só é possível utilizar nas as ferramentas de manipulação, *move*, *rotate* e *scale*, nas UV's ou em UV Shell que é um conjunto de UV's.



b. Automatic mapping

Gera um mapa UV automático de um objeto, verificando para que lado determinado conjunto de face está predominantemente voltado e separa as partes gerando o mapa, este tipo de mapa é indicado apenas para objetos extremamente simples, já que não há um bom controle de onde será feita a divisão ou tamanho das peças.

c. Planar mapping

Gera uma única projeção a partir de um eixo que pode ser selecionado em suas opções, pode ser aplicado em um grupo específicos de faces criando um UV Shell, é importante escolher o eixo correto para fazer a projeção e manter a caixa **keep image width/height ratio** para que não aconteçam deformações na UV Shell.

d. Cylindrical e spherical mapping

Aplicam projeções em forma de cilindro e esfera, são indicadas para formas específicas que se encaixem nessas duas formas, especialmente o cilindro. Assim como automatic mapping é mais indicado em objetos mais simples.

04. UV toolkit

Existe um conjuntos de ferramentas que permitem uma melhor manipulação da UV, garantindo uma UV coerente e com melhor aproveitamento de imagem. No caminho **UV/UV editor/tools/ show UV toolkit**, é habilitado uma janela com diversas ferramentas úteis, organizadas em subgrupos.

a. Transform

Ferramentas voltadas para mover, escalar e rotacionar UV's, além de fazer edições em pivôs de transformação, em suas opções podemos destacar.

»Flip:

Inverte a imagem na horizontal ou vertical, é normalmente utilizada quando uma *UV Shell* está invertida mostrando a imagem, especialmente texto, de forma espelhada.

»Texel density:

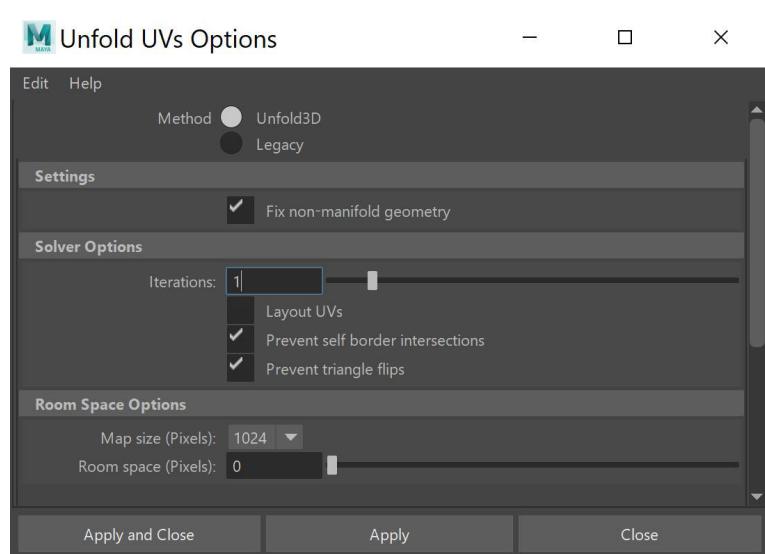
Transfere a densidade de pixel de uma *UV Shell* para outra, fazendo que as duas tenham a mesma resolução, ou seja, o mesmo tamanho no *UV map*, basta dar o comando **get** na peça que quer manter o tamanho e **set** nas peças que quer alterar.

b. Cut and sew

Agrupa ferramentas que unem ou separam *UV Shell*, sendo suas principais ferramentas **cut** que corta a *UV Shell* nas edges selecionadas e **sew** que une as edges selecionadas.

c. Unfold

Umas das mais importantes ferramentas no *UV toolkit*, quando geramos projeções através de *planar maps* ou outros *mappings*, as UV's criadas muitas vezes então desproporcionais as faces do objeto devido a uma inclinação da face em relação a projeção, o **unfold** “desdobra” estas UV's tentando deixá-las o mais próximo da proporção real das faces evitando distorções dos pixels representados nas mesmas.



O modo **Unfold 3D**, tende a desdobrar com mais eficiência estas UV's em relação ao objeto tridimensional, você pode verificar se este modo está ativado nas opções de **UV/UV Editor/Modify/Unfold**, Caso este modo não esteja disponível neste caminho será necessário ativar o plugin no caminho:

Windows/settings references/plug-in manager/Unfold3D.mll

d. Align and snap

Ferramentas que ajudam a posicionar e alinhar os *UV Shell*, permite organizar o mapa de UV de forma personalizada. **Align** permite que alinhar *UV Shells* em diversos eixos ou pivôs de posicionamento, já o comando **Snap** permite o posicionamento em um dos nove quadrantes do quadrado destinado ao mapa UV, também existe o comando **Snap Together** que empilha *UV Shells* a partir de uma UV.

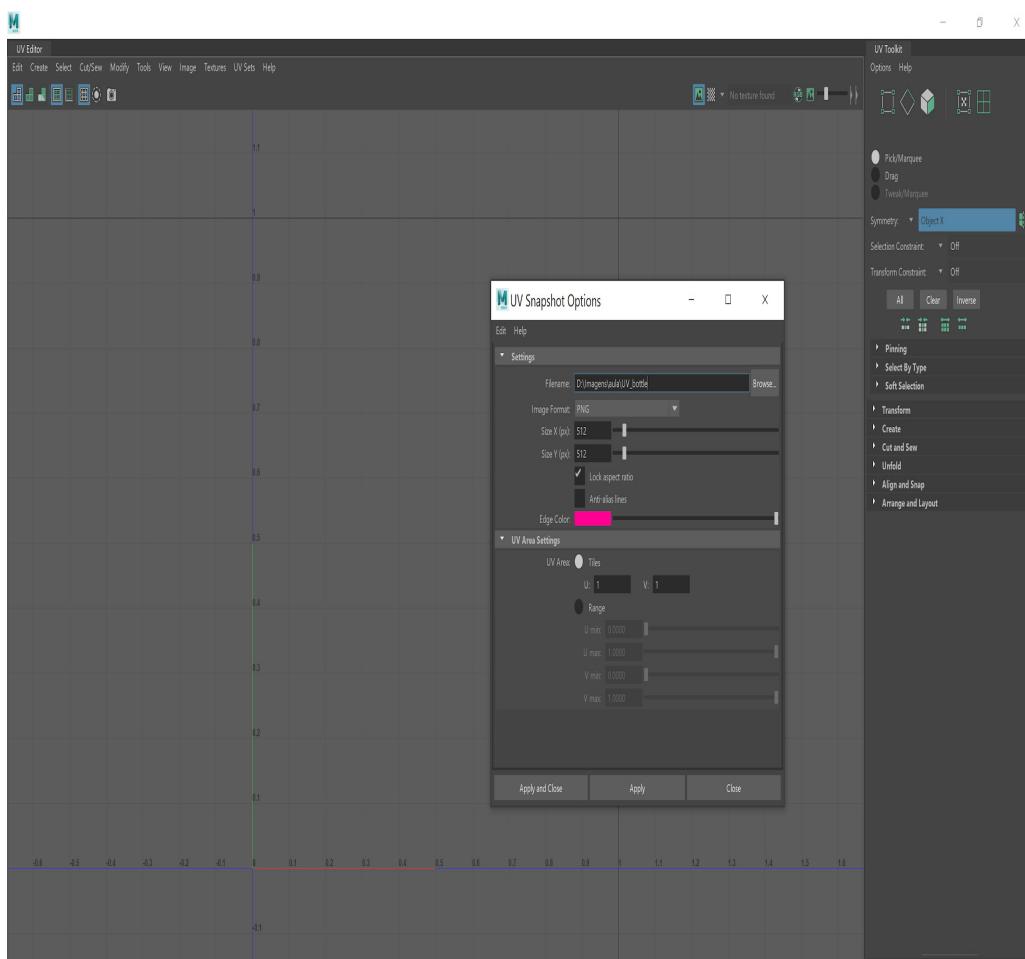
e. Arrange and Layout

As ferramentas deste menu ajudam a organizar automaticamente os *UV shells*, procurando ocupar da melhor forma possível o mapa de UV. **Orient shells** rotaciona o *UV Shell* de forma a deixar seus limites alinhado em ângulos de noventa graus as deixando "retas", a ferramenta **Stack Shell** empilha as *shells* selecionadas e a **Stack Similar** empilha as *shells* que tem a mesma topologia. O comando **Layout** move rotaciona e escala os *UV Shells* para maximizar a utilização de espaço dentro do quadrante de uv, organizando todo o mapa UV.

f. UV snapshot

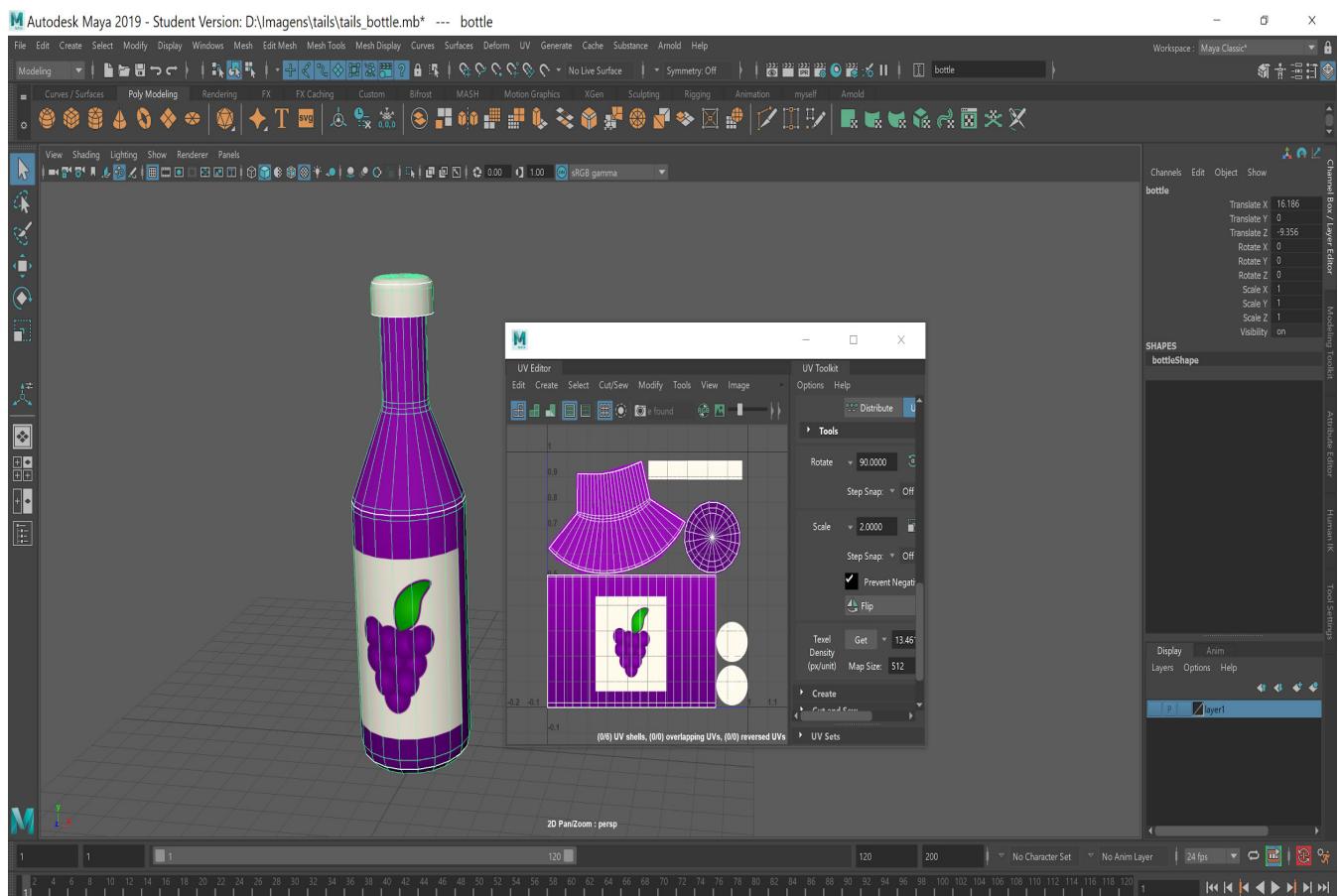
É possível gerar uma imagem do mapa UV de um objeto selecionado no caminho **UV/UV editor/image/UV snapshot**, é gerada uma referência que podemos utilizar para criar uma textura em qualquer editor de imagem.

No *UV snapshot editor* podemos configurar onde a imagem será salva, a resolução da imagem, o tipo de imagem gerada e a cor da representação da UV.



EXERCÍCIO EXTRA 5

Modele um objeto simples como uma garrafa ou jarro, prepare seu mapa UV, ferramentas como, **planar mapping, cylindrical mapping, unfold, texel density e layout** serão muito importantes. Depois use o **UV snapshot** para gerar uma imagem, então faça uma pintura em um editor de imagens a sua escolha.



Com este exercício aprendemos:

- » Criar mapas UV.
- » Utilizar o mapa UV para orientar a criação de uma textura.
- » aplicar texturas em nossos modelos.

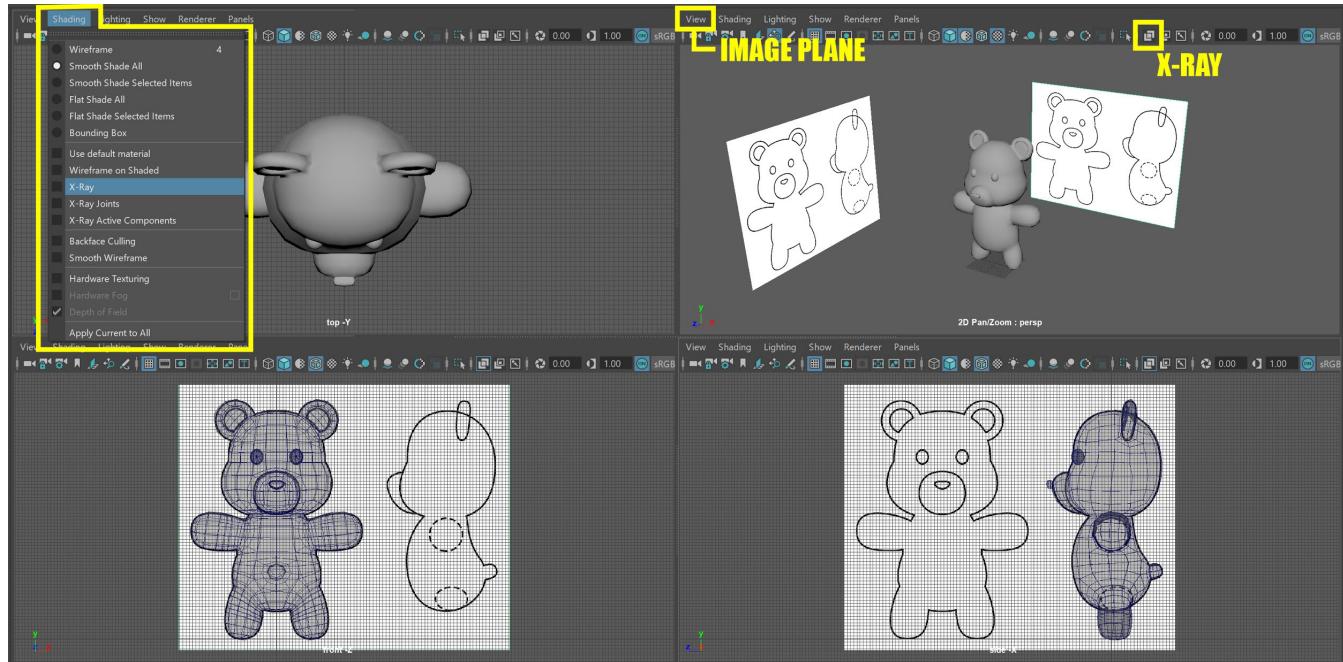
Salve a imagem em **JPEG** ou **PNG**.

MODELAGEM ORGÂNICA

A modelagem orgânica, quando as formas são mais livres, menos geométricas, requer prática, o fluxo de muitos artistas na modelagem de personagem normalmente seguem em ferramentas mais voltadas para esse fim, no entanto é possível atingir resultados profissionais utilizando o Autodesk Maya. As principais preocupações quanto a este tipo de modelagem é quanto a topologia (formato do wireframe) deixando a malha leve e adequada para animação, neste tópico abordaremos algumas práticas que podem facilitar este processo.

01. Model sheet

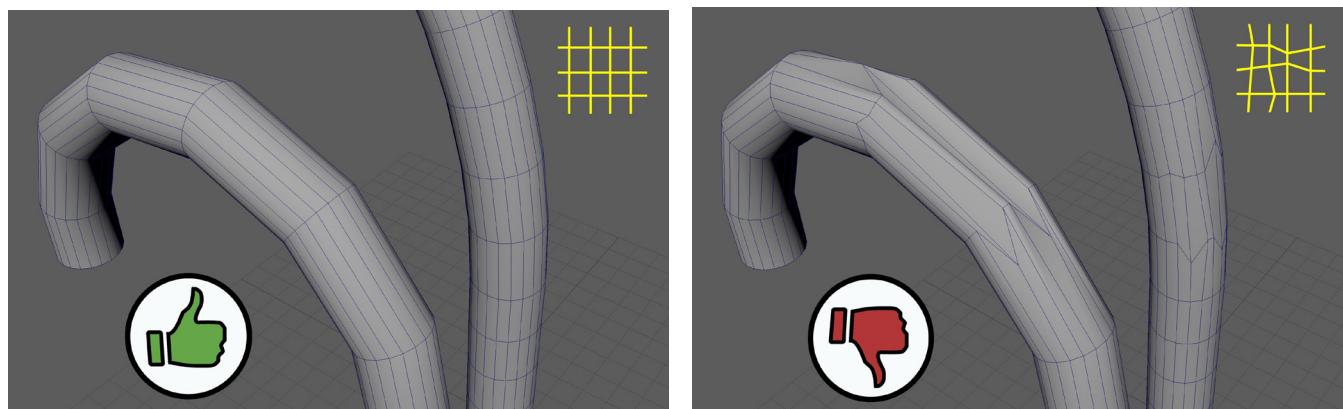
É uma imagem utilizada para demonstrar proporções e detalhes de um determinado personagem, na modelagem 3D serve como referência para posicionar a malha dando formato as partes específicas. Um model sheet de personagem próprio para modelagem normalmente traz a figura em posição neutra e em diversos ângulos. Para aplicar um model sheet deve se utilizar o caminho **view/image plane/import image**.



As imagens devem ser aplicadas nas viewport ortográficas (*side, front ou top*), e no channel box é possível configurar seu tamanho, posição da forma adequada para utilização. É importante prestar atenção na escala e posição das imagens que devem estar alinhadas entre si e com o centro do mundo, para visualizar a imagem por trás da malha use o comando **shading/X-ray**.

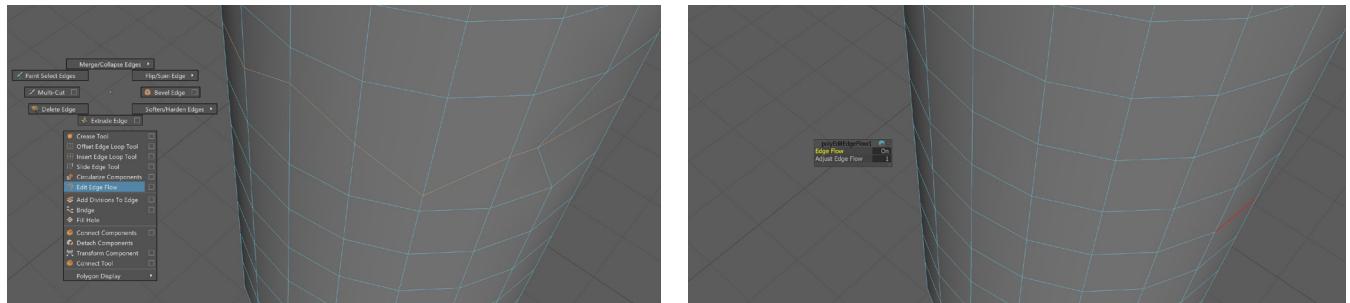
02. Fluxo de malha

Quando modelamos devemos cuidar do formato geral do wireframe, por mais que uma determinada malha tenha o formato desejado, sem um bom fluxo que mantenha uma “grade coerente, ela pode ser renderizada inadequadamente gerando sombras e quinas rígidas e caso esta malha seja animada, pode dobrar de forma estranha com formato indesejado.



a. Edit edge flow

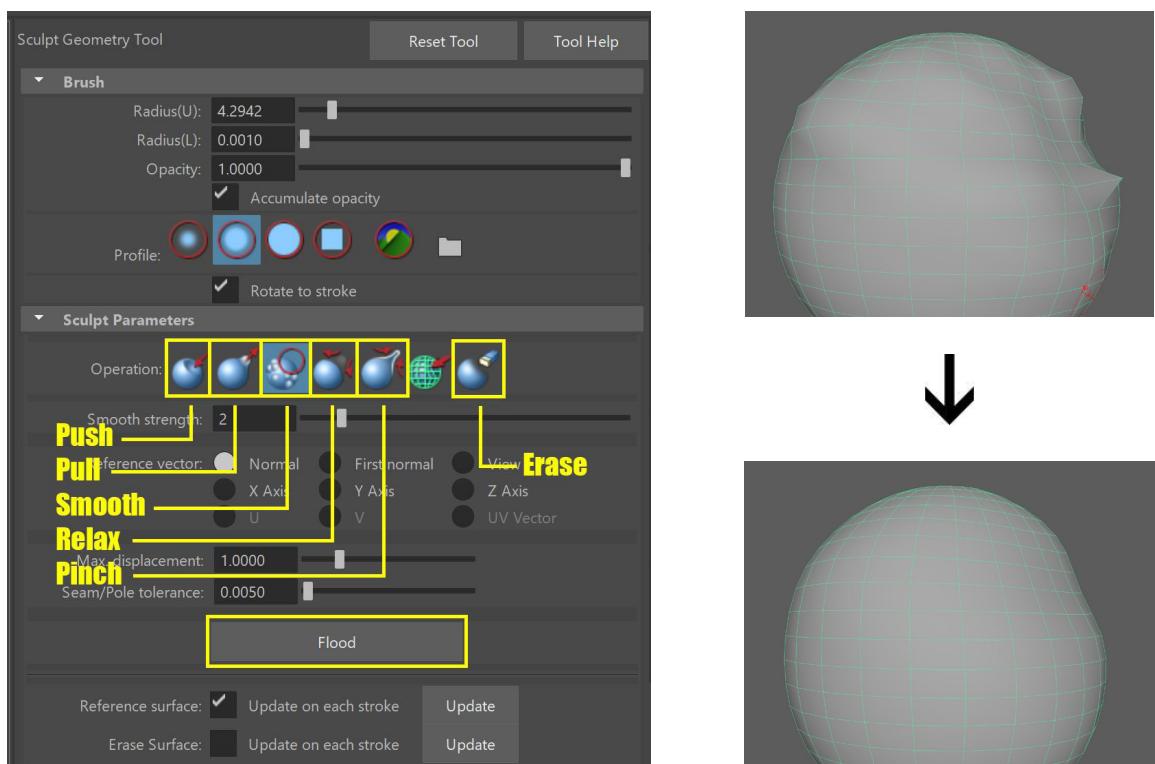
Uma forma de organizar o fluxo da malha é a ferramenta **Edit edge flow**, ela alinha os vértices de forma a deixar o formato mais orgânico, por vezes a circunferência do looping é ampliada mas isto pode ser consertado com a ferramenta scale.



b. Sculpt geometry tool

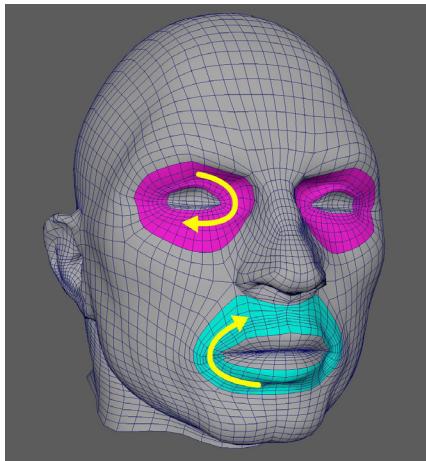
Esta ferramenta é um pincel que pode alterar partes da malha puxando empurrando ou reorganizado os vértices, no caminho **Surface/Sculpt** geometry tool esta *brush* é ativada, no menu **tool settings** estão as configurações exatas desta ferramenta. Cada opção do pincel afeta a malha de uma determinada maneira, **Push** empurra, **Pull** puxa, **Smooth** suaviza os vértices deixando homogênea a distância entre eles, **relax** relaxa os pontos de maior contraste entre os vértices, **Pinch** aproxima os vértices em direção ao centro do pincel e **Erase** desfaz as alterações feitas pelo sculpt geometry.

Uma forma interessante de trabalhar com a *sculpt geometry* é utilizando o botão **Flood**,



Essa ação aplica a deformação em todo objeto, ou nos vértices selecionados, principalmente nas opção *Smooth* e *Relax*, ajustando o fluxo de malhas de forma automática, usando a ferramenta dessa forma normalmente a malha perde um pouco de volume, mas basta reequilibrar as escalas e posições em pontos específicos.

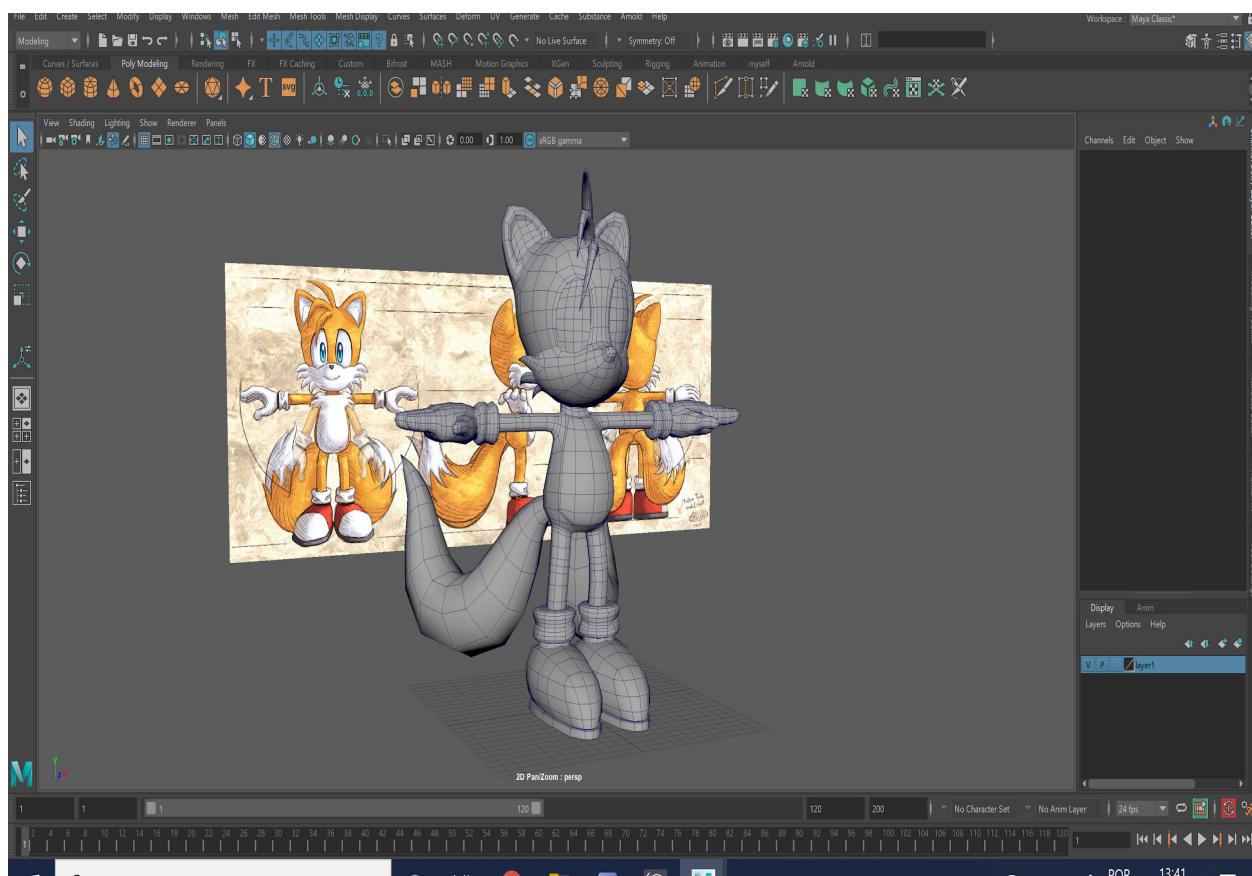
03. Looping de animação



Uma preocupação importante a se levar em consideração ao criar um modelo, especialmente caso este tenha que ser animado, são os *loopings* de animação, este fluxo permite a deformação correta da malha, é importante evitar triângulos próximos a lugares com muita mobilidade. Aumentar a quantidade de loopings em partes que se dobram tais como: cotovelos, joelhos ou dobras dos dedos, partes expressivas como olhos e boca também precisam de loopings extras em volta deles para facilitar alterar seus formatos.

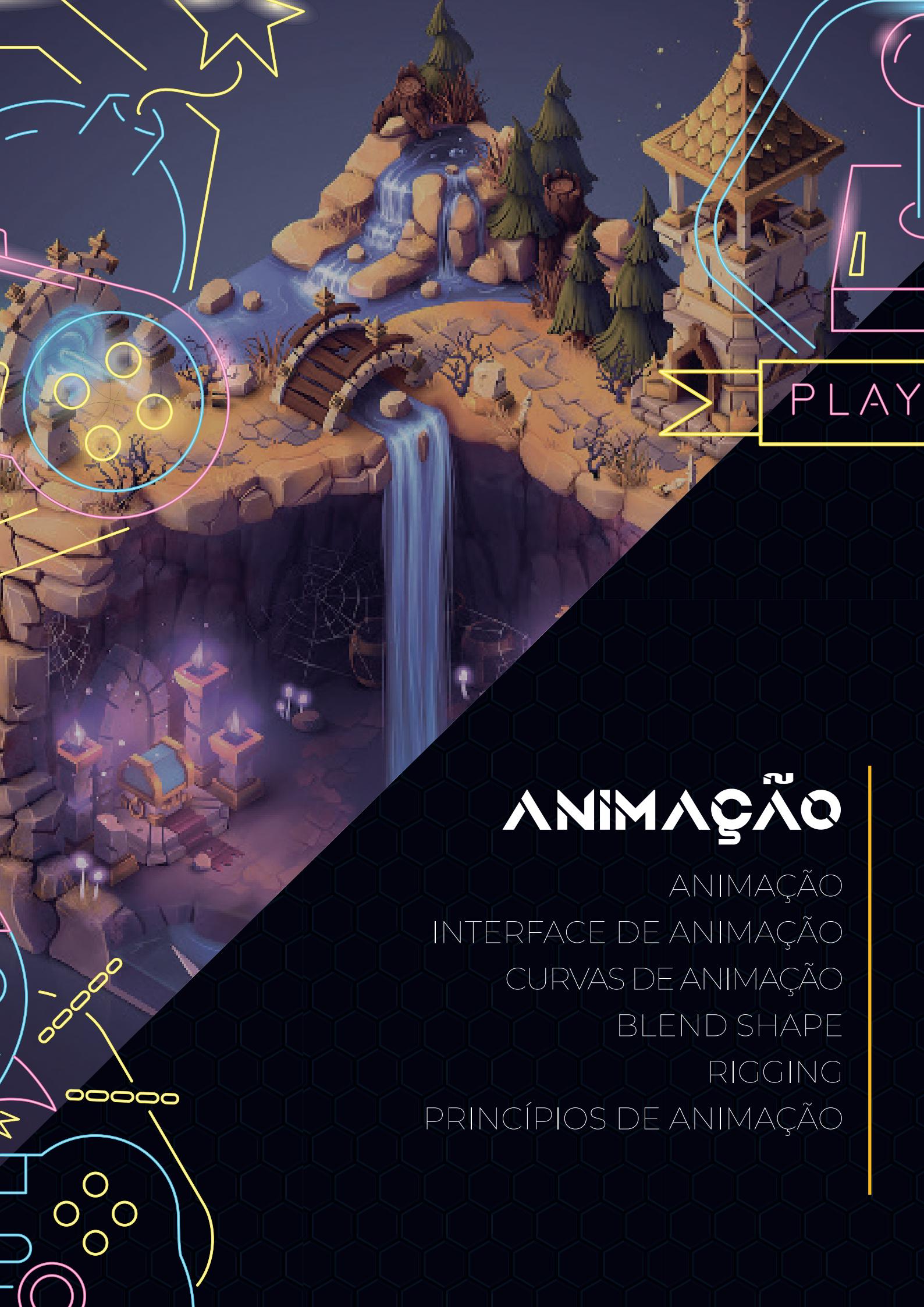
EXERCÍCIO EXTRA 6

É hora de colocar em prática tudo que aprendeu. Escolha um personagem que te agrade e faça uma modelagem dele, analise qual ferramenta pode te ajudar em cada etapa.



É muito recomendado procurar várias imagens de referência para compreender melhor o formato do personagem, se preocupe com o fluxo de malha. Trate esta atividade como uma oportunidade de começar ou melhorar seu portfólio.

A modelagem assim como várias outras atividades no design requer, prática e dedicação para serem aprimorados, criar vários modelos de diferentes tipos e estilos, ajuda a memorizar as ferramentas e desenvolver conhecimento em volumetria, estudar anatomia também é algo que pode ajudar a desenvolver suas habilidades, é importante sempre se manter estudando e se atualizando para ser um bom profissional.



ANIMAÇÃO

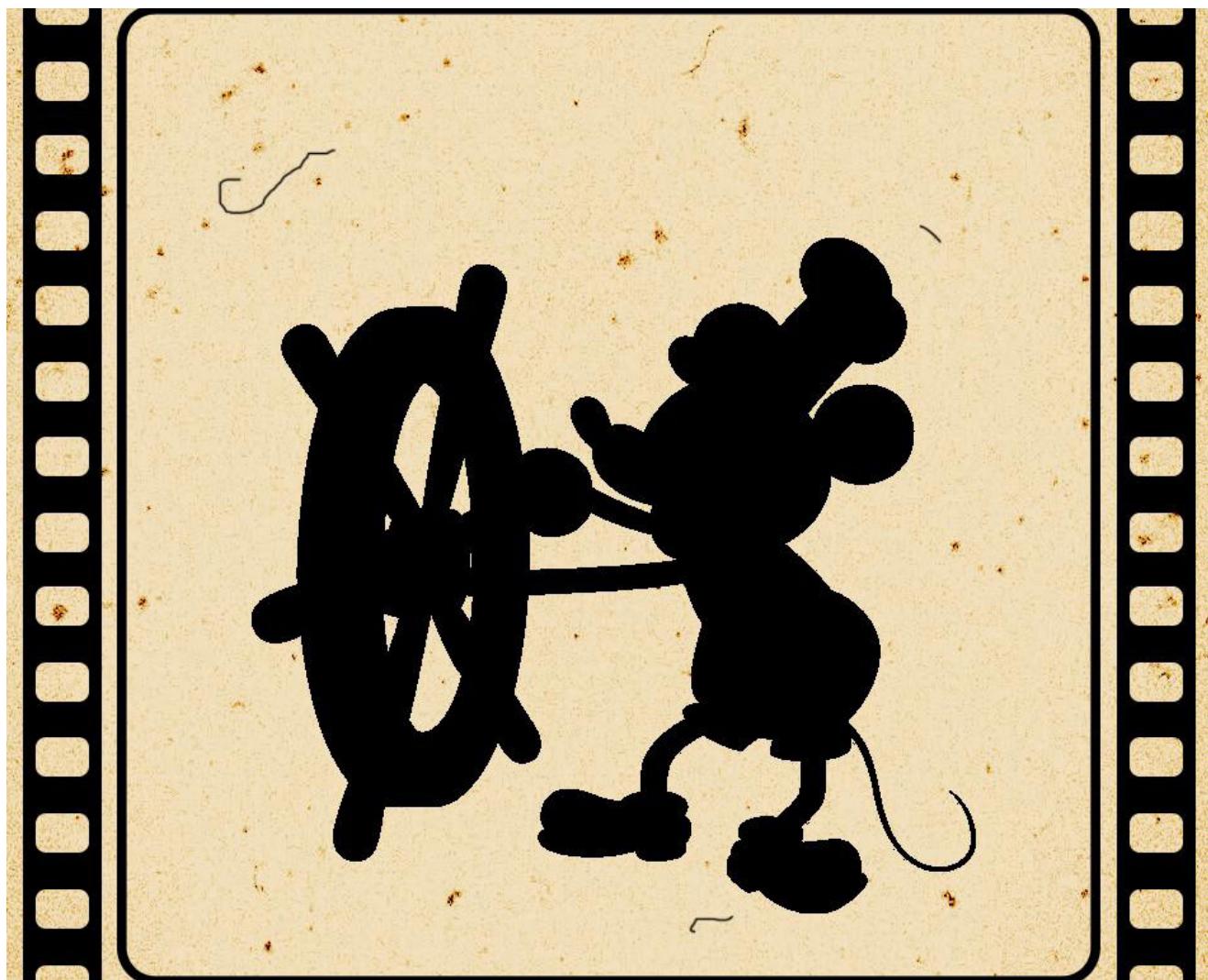
ANIMAÇÃO
INTERFACE DE ANIMAÇÃO
CURVAS DE ANIMAÇÃO
BLEND SHAPE
RIGGING
PRINCÍPIOS DE ANIMAÇÃO

INTRODUÇÃO

A animação, esta forma de expressão que tem uma capacidade enorme de cativar, é uma das áreas mais empolgantes do design, costuma tornar tudo mais chamativo, principalmente quando comparamos algo com movimento, com algo estático, muitas vezes dizemos que este elemento “ganhou vida”, e esta analogia não é por acaso, já que a palavra animação tem sua origem no latim, anima, que significa alma.

As animações estavam lá no início, nasceram praticamente em conjunto com tudo o que viria a se tornar o audiovisual, alguns diriam até mesmo que, de certa forma, vieram antes disso tudo. de uma distração nas salas de cinema, passando antes da “atração principal” as mega produções de hoje, as animações estão aqui para ficar, os meios de produção e distribuição mudam, mas tem seu lugar garantido. Obviamente os jogos também sempre andaram de mãos dadas com a animação, vencendo desafios desde sempre “como representar movimento com uma animação de dois frames?” e agora dispõe recursos de ponta como captura de movimentos e aplicação de física de forma realista, mas ainda assim os realces e retoques do animador se mantêm importantes.

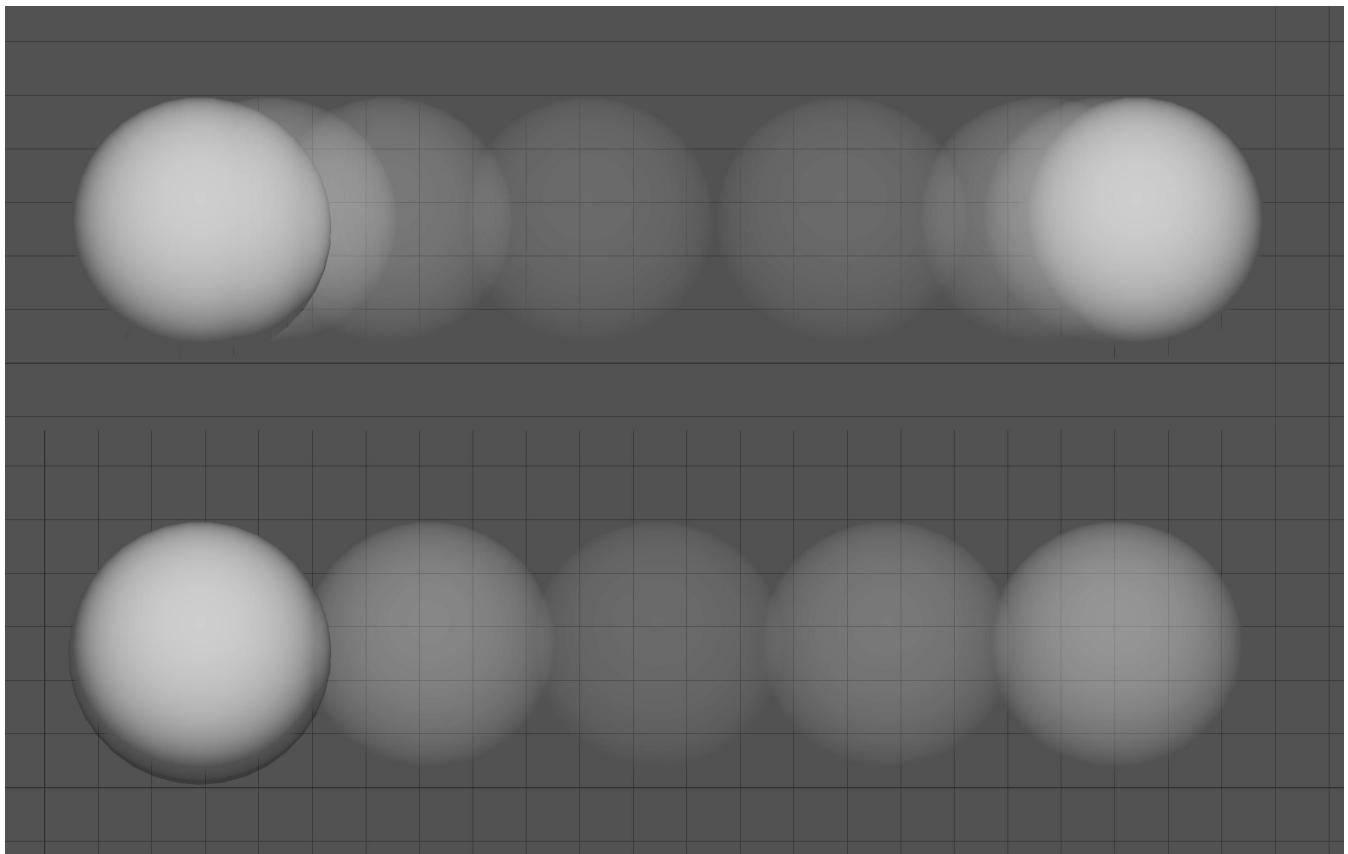
Neste módulo vamos aprender o necessário como preparar os modelos para serem animados e como produzir e exportar estas animações 3D do autodesk Maya para utilizarmos em jogos, mas não apenas utilizar as ferramentas para isto, mas estudar conceitos sobre o que torna as animações convincentes e carismáticas.



ANIMAÇÃO

Por mais incrível que pareça, o ato de animar é algo simples, resumindo de forma grosseira, uma animação nada mais é que a sucessão de imagens que se alternam de forma gradual para passar a ideia de movimento, no entanto, se fazer uma animação acontecer é algo simples, passar verdade e beleza a este movimento não é tão fácil, exige prática e uma boa noção de tempo e espaço.

A forma como visualizamos as animações, é semelhante em diversas mídias, um filme, animação 2D, 3D, um flip book ou um jogo, são visualizados por imagens estáticas que se sucedem em uma cadência rápida, em mídias digitais chamamos isto de **FPS (Frames per Seconds)**, ou seja, quantas imagens são exibidas no intervalo de um segundo, quanto mais imagens expostas nesse período de tempo, mais fluido o movimento se mostra, e o que passa a ideia de que algo se move de forma rápida ou lenta é o espaço que algo se desloca entre um quadro e outro.



No Maya a animação é feita através da interpolação, um processo onde se salva as coordenadas do objeto em um frame e depois em uma segunda posição em outro, o programa calcula as poses intermediárias em cada quadro de acordo com a taxa de frames. por padrão acrescenta aceleração entre os frames não os deixando uniformes, o que dá mais naturalidade a animação, mas obviamente a intensidade desse efeito pode ser ajustada e em personagem mais complexos precisamos de uma preparação mais profunda para que o modelo se move corretamente, chamamos esta preparação de rigging, onde normalmente se cria um esqueleto para articular o modelo.

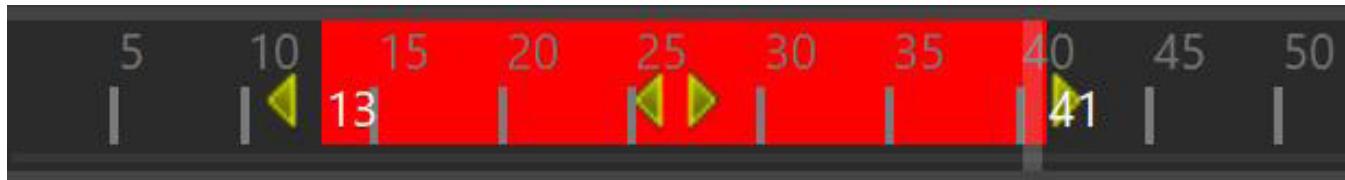
INTERFACE DE ANIMAÇÃO

O autodesk Maya tem diversos recursos de animação, os principais se agrupam na **timeline**, essencialmente opções de criar editar e visualizar *frames*, que são coordenadas armazenadas em um determinado momento na *timeline*, cada objeto tem seus próprios *frames* que são marcados por traços vermelhos na *timeline* quando o selecionamos.



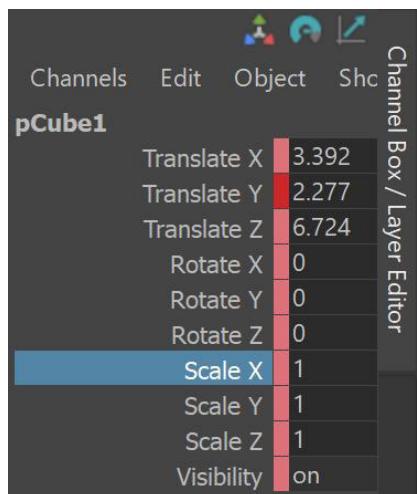
01. Determina quantos frames existem e estão sendo visualizados, o primeira caixa diz onde a animação começa, a segunda e a terceira dizem onde se inicia e se finaliza a visualização dos *frames*, para que possa ver apenas uma parte isolada da animação, e a quarta caixa mostra onde a animação termina, a informação pode ser digitada nestes espaços. A barra no meio é um controle manual destes parâmetros.
02. O *frame* atual fica marcado em cinza claro na *timeline*, é possível escolher um *frame* específico com o mouse.
03. Os **Keyframes** (coordenadas salvas em um quadro) são marcados com um traço vermelho, para criar um keyframe basta fazer as alterações desejadas e pressionar **S**.
04. Esta caixa sinaliza o *frame* atual, é possível digitar o quadro que deseja exibir.
05. Player da cena, toca a animação avança quadros, inverte o movimento, de acordo com os comandos.
06. Abre a janela **Preferences**, referente às preferências de animação, nela é possível alterar configurações do player e dos frames. Uma dúvida frequente é sobre a velocidade da animação acelerada, para alterar isto é necessário ir em **Preferences/Time Slider/Playback speed** e mudar a opção **Play every frame** para a velocidade com base no FDS escolhido para animação.
07. **Auto Key**, quando ativado, grava automaticamente um keyframe sempre que um atributo de um objeto com animação é alterado. Agiliza e refina o processo, mas requer atenção redobrada quando se manipula os objetos.
08. **Looping** do *playback*, controla a repetição da animação, tem as opções de repetir infinitamente, três vezes ou tocar a animação apenas uma vez.
09. **FPS**, taxa de quadros por segundo da *timeline*, o ideal é configurar esta opção antes de começar a animar (normalmente 60fps).

Para selecionar um frame ou um intervalo na timeline basta fazer a seleção com **Shift** pressionado e a área será sinalizada com vermelho, sendo possível mover pelas setas centrais ou escalar a seleção pelas setas das pontas.



Clicando com o botão direito do mouse sobre um frame ou intervalo selecionado na *timeline*, as opções de **cut, copy, paste e delete** podem ser aplicadas, funcionando de forma semelhante ao que estamos acostumados com textos, já o Paste connect, cola os frames copiados, mas leva em consideração a pose atual do objeto criando continuidade, o comando **Snap** ajusta um *keyframe* que esteja entre dois *frames* de uma animação, o que normalmente acontece quando um intervalo de frames é escalado.

Os *key frames* também podem ser manipulados utilizando o *channel box*, quando um atributo tem um *key* recebe uma marca vermelho claro após seu nome, e quando estamos com o momento da *timeline* onde há um *keyframe* ele marca com vermelho vivo. Podemos criar um novo *key* clicando com o direito sobre o atributo e escolhendo **Key selected**, também é possível usar os comandos **cut, copy, paste e delete** em atributos específicos no *channel box*, no entanto quando esses comando são feitos no *channel box* todos os *key* deste atributo são afetados.

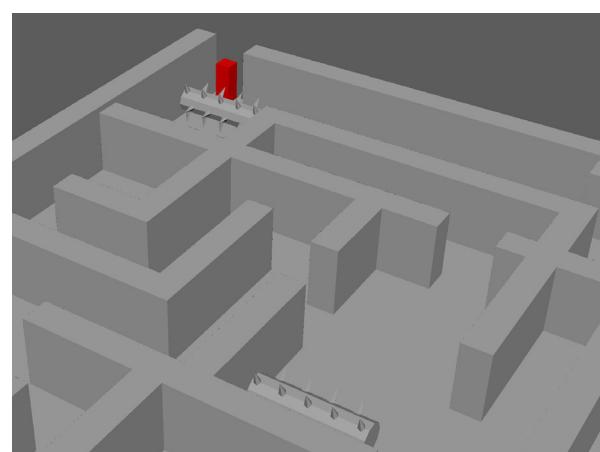


EXERCÍCIO I

Faça uma animação fazendo uma primitiva percorrer por um espaço específico, desviando e interagindo com outros objetos animados.

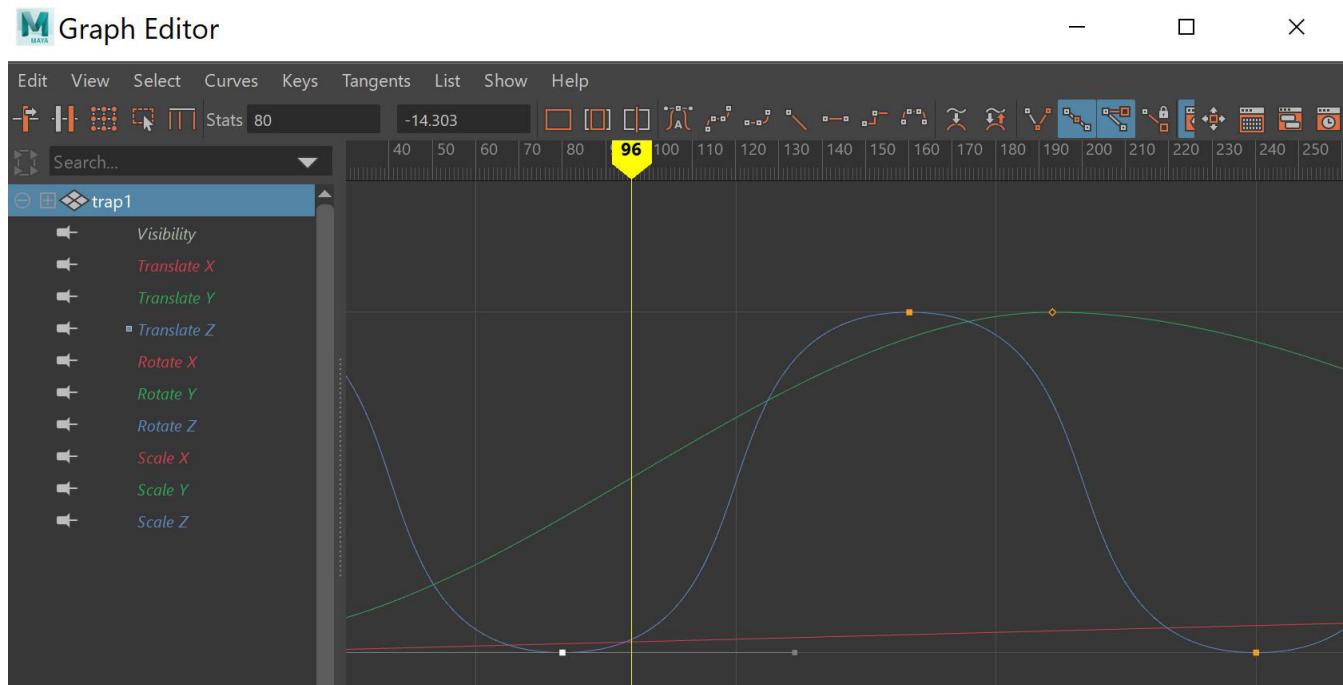
Com esse exercício aprendemos:

- » Criar e organizar *keyframes*
- » Lidar com animação de múltiplos objetos
- » lidar com a *timeline*



CURVAS DE ANIMAÇÃO

Quando criamos um *keyframe*, o Maya acrescenta pesos que fazem com que a interpolação tenha aceleração e desaceleração em seu movimento, esses pesos podem ser visualizados e editados nas curvas de animação, no caminho **windows/animation editors/graph editor**, nesta janela se tem acesso a todos atributos, seus *keyframes* são conectados formando as curvas de animação.

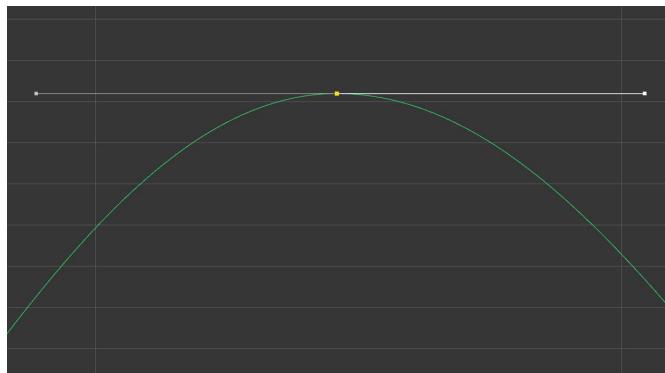


No quadrante da direita há uma lista com os objetos selecionados e seus atributos, é possível visualizar apenas atributos específicos os selecionando nesta lista, no quadrante da esquerda há uma gráfico onde na vertical temos os valores dos atributos e na horizontal os frames da *timeline*, os *key* estão aqui representados por pequenos quadrados laranja conectados por uma linha da cor atrelada a seu eixo de transformação. A navegação de sua câmera funciona de forma semelhante a uma *viewport* ortográfica, seus comandos básicos são:

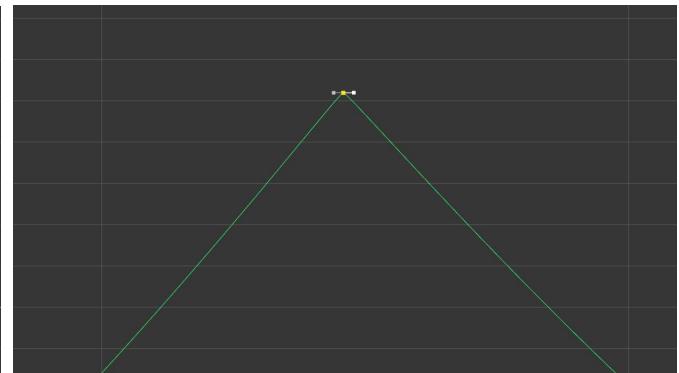
- » **Translate (W):** Mover *keyframes* selecionados alterando o valor do atributo ou sua posição na *timeline*.
- » **Scale (R):** Escala *keyframes* aumentando ou diminuindo a distância entre eles de forma proporcional, de acordo com o ponto onde se clica.
- » **Frame (F):** focaliza dando zoom nos objetos selecionados, é importante para visualizar de forma correta o desenho da curva de animação.
- » **All (A):** mostra por inteiro as curvas de animação selecionadas.
- » **Shift:** ajuda a mover a seleção em linha reta, na vertical ou na horizontal de acordo com o primeiro movimento aplicado.
- » **Delete:** deleta os *frames* selecionados no *graph editor*.

01. weight tangent

Quando criamos Keyframes, automaticamente é feita a interpolação entre eles, neste processo o movimento simula um pouco de física, de forma que os movimentos ao invés de mudar direção de forma brusca, pareçam ter um pouco de aceleração ou desaceleração, esta variação chamamos de pesos, estes podem ser manipulados nas curvas de animação aumentando ou diminuindo o efeito de acordo com a necessidade, para habilitar esta edição de forma livre é preciso seguir o caminho **Curves/Weighted tangents**, dentro do graph editor, agora as tangentes podem ser aumentadas ou diminuídas, basta selecionar um dos lados da tangente e o mover com o botão do meio do mouse, pressionando **Shift** se altera o tamanho da tangente sem alterar sua direção.

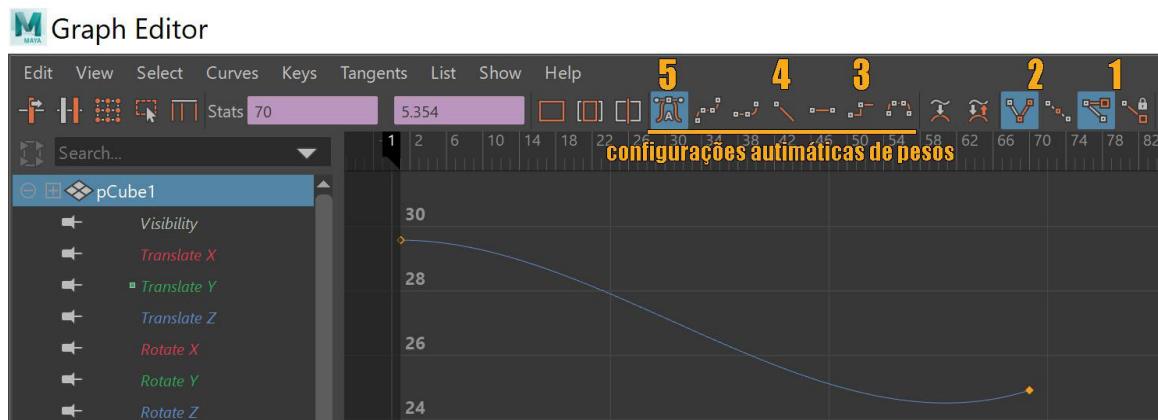


Tangentes grandes mudanças suaves



Tangentes pequenas mudanças bruscas

Comandos relevantes ligados às tangentes:



Free/lock tangent length: Permite que se bloquee ou libere o aumento de pesos de um keyframe específico.

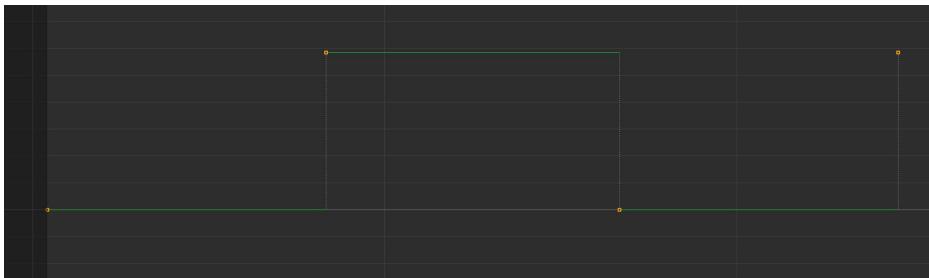


Break/Unify tangents: Torna possível trabalhar as tangentes de um keyframe de forma separada ou a juntá-las novamente, por vezes um movimento pode ter a entrada intensa e sair suave ou o contrário, esse comando permite que isto aconteça.

Existem a opções de ajuste automático dos pesos, que configuram as tangentes de forma a conseguir resultados diferentes, as mais contrastantes são:



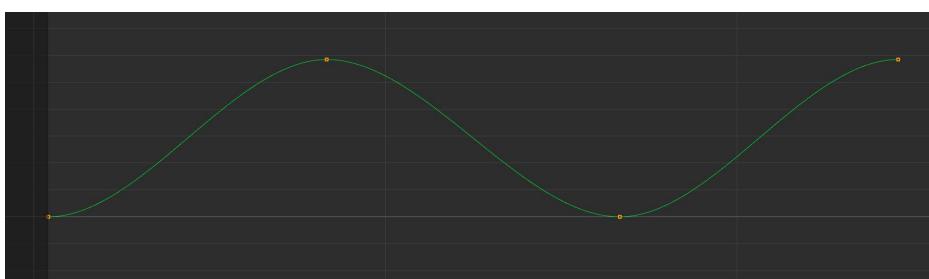
Step tangents: Esta configuração de tangentes mantém o objeto parado até a chegada de outro keyframe, não havendo nenhuma interpolação, utilizadas em movimentos muito bruscos ou rápidos, como um teletransporte por exemplo.



Linear tangents: Deixa o movimento constante, sem aceleração, indicados para movimentos mecânicos ou loopings simples, como exemplo de engrenagens girando perfeitamente.



Auto tangents: Configuração padrão das tangentes, é um ajuste neutro, especialmente útil para fazer um refazer pesos básicos depois que *frames* são movidos de lugar na *timeline*.



EXERCÍCIO II

Faça uma esfera simular o efeito de uma bola quicando no chão de forma realista, represente detalhes como o movimento ter uma mudança suave nos saltos e brusca no impacto com o solo, use as curvas e seus pesos no graph editor, para conseguir este efeito e ajustar a gradação da altura dos saltos:

Com este exercício aprendemos:

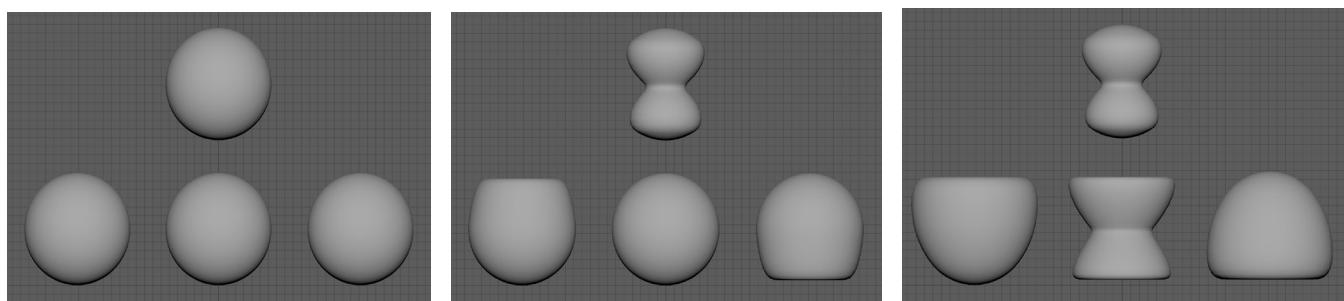


- » Fazer ajustes de posicionamento pelo graph editor
- » Editar separadamente os atributos
- » Trabalhar tangentes para ajustar a aceleração
- » Simular movimentos convincentes

BLEND SHAPE

Os Blend Shapes são atributos que podemos criar em objetos, eles armazenam transformações em vértices gerando uma animação vinculada a estes atributos. Seu uso mais comum é dar expressão a personagens, já que vários atributos podem ser vinculado a um único objeto, vinculando pontos diferentes como boca, olhos, sobrancelhas o que deixa as expressões bem variadas, outra vantagem desta técnica é que deformações leves são feitas de forma mais fácil por transformações nos vértices.

Para aplicar o *blend shape* o objeto deve ser duplicado (**Ctrl + D**) e as alterações devem ser feitas na cópia, com os objetos selecionados, o efeito é aplicado em **deform/blend shape**, é adicionado um atributo que varia de um a zero em que a forma pode ser alternada do formato original para a modificada gradativamente.



Múltiplas cópias podem ser utilizadas simultaneamente em um mesmo *blend shape*, ao selecionar os objetos o último selecionado será o que vai receber o atributo, os nomes atribuídos aos objetos modificados serão adicionados como subatributos, é interessante trabalhar separadamente partes do modelo de acordo com a necessidade.

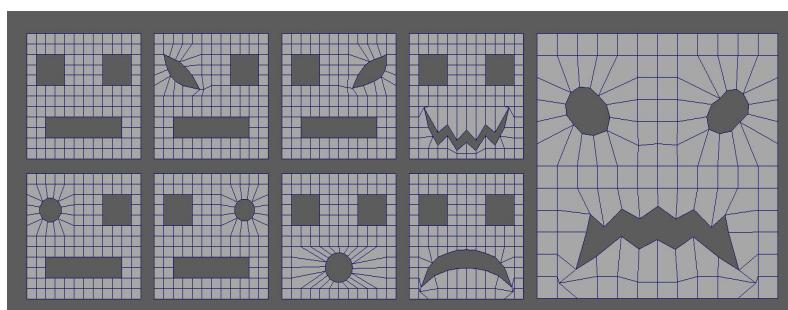
INPUTS

blendShape1

Envelope	1
UP	1
CENTER	1
DOWN	1

EXERCÍCIO III

Faça um rosto neutro utilizando um plano subdividido, faça várias expressões neste plano usando *blend shape*, trabalhe separadamente cada olho e a boca para aumentar a versatilidade de expressões, nomeie de forma coerente cada cópia para organizar o atributo criado.

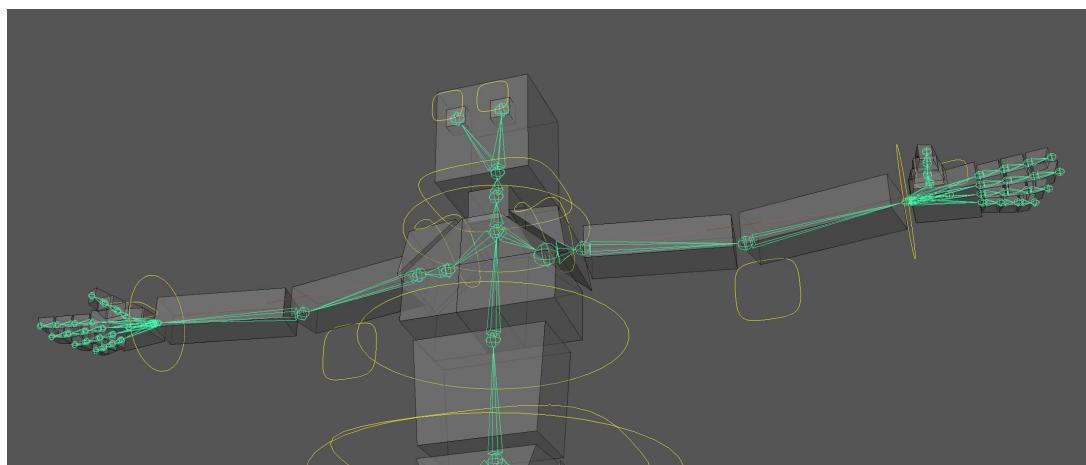


Com este exercício aprendemos:

- » Usar blend shape para trabalhar expressões.
- » trabalhar múltiplas transformações simultâneas.
- » organizar os subatributos do *blend shape*.

RIGGING

Ao animar personagens complexos, precisamos preparar os modelos para este movimento, articulando a malha para tornar mais fácil e rápido processo, chamamos este trabalho de rigging, que tem várias etapas que requerem determinado esforço, mas depois de prontas automatizam de forma bastante prática os movimentos. As principais etapas do Rigging são a criação do esqueleto, que determina os pontos de articulação que dobram a malha, a pintura de pesos de define como exatamente a malha reage ao esqueleto e por fim a construção dos controles que manipulam os ossos. Vale lembrar que as opções de *rigging* aparecem quando se muda a caixa de *modeling* para *rigging*.

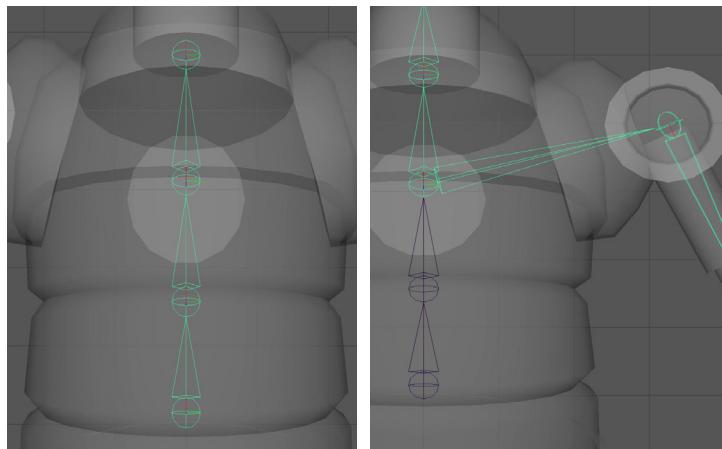


01. Joints

A primeira etapa ao fazer o rigging de um personagem é criar seu esqueleto, esta estrutura é feita por elementos que chamamos de Joints, são pontos de transformação semelhantes a pivôs, normalmente formam uma hierarquia de parentesco se conectando a outros joints, a ferramenta mais básica para criação é ***skeleton/create joint***,

A ferramenta cria *joints* a cada clique com o botão esquerdo do mouse, o próximo *joint* será filho do anterior, **delete** retrocede um passo e **enter** finaliza a hierarquia. Para criar uma ramificação, basta usar a ferramenta em cima de um *joint* de uma hierarquia já existente.

É importante criar *joints* nas vistas ortográficas ou utilizando *snaps*, já que a perspectiva costuma não dar bons resultados com esta ferramenta.

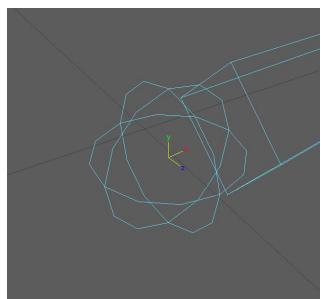
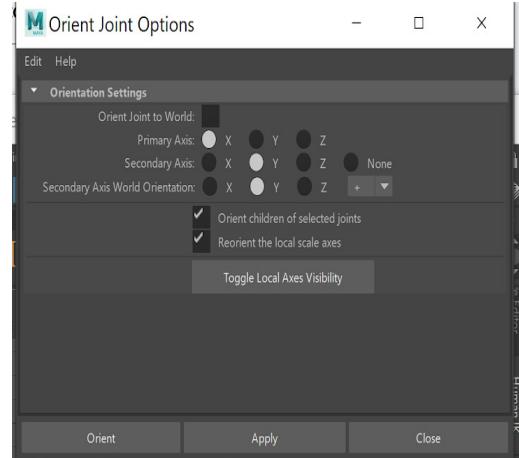


Outras ferramentas dentro de *skeleton* que trabalha com joints:

- » **Insert joint:** acrescenta um *joint* entre dois joints já existentes, bastando clicar no pai e arrastar.
- » **Remove joint:** deleta o *joint* selecionado, sem deletar seus filhos.
- » **Connect joint:** conecta um *joint* selecionado a uma hierarquia, a operação **parente (P)** tem efeito aproximado.
- » **Disconnect joint:** divide uma hierarquia do a partir do ponto selecionado.
- » **Reroot skeleton:** Torna o *joint* selecionado o novo **root (pai de toda a hierarquia)**

a. Orient joint

Boa parte das transformações feitas no esqueleto vai ser feita a partir de rotação, para que tudo ocorra da forma desejada é importante dar atenção a orientação dos joints, normalmente o eixo principal é o X, que aponta para o joint anterior, a ferramenta que faz esta orientação automaticamente é Skeleton/Orient joint. Também é possível fazer a orientação dos joints se alinhar com os eixos do mundo, o que normalmente é utilizado para juntas mecânicas.

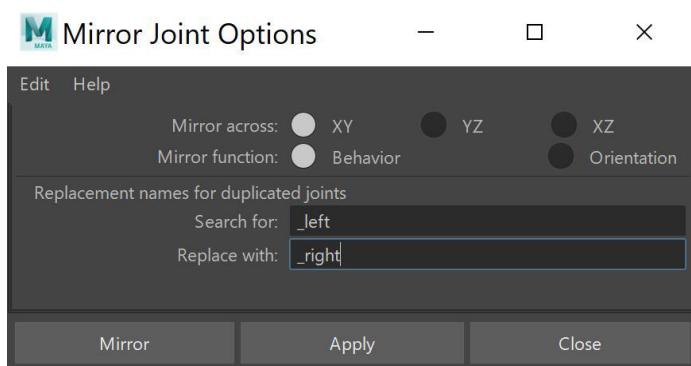


Por vezes é necessário fazer a orientação manual de certos pontos do esqueleto, para rotacionar o eixo de rotação de um joint, é preciso alterar a forma de seleção para **component type (1)** e ativar a opção **miscellaneous (2)**, agora pode-se selecionar e rotacionar o eixo com a ferramenta rotate.

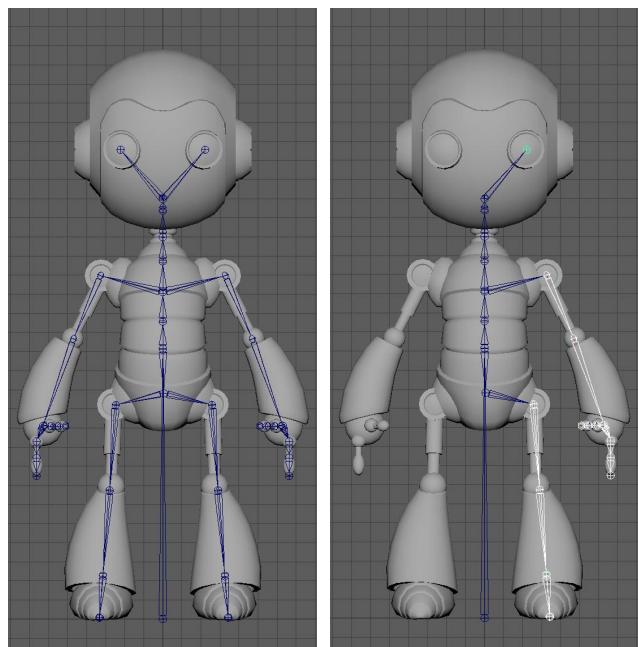


b. Mirror joint

Assim como fazemos com a malha, podemos espelhar nossos ossos, o que agiliza e garante a simetria do trabalho, no atalho **Skeleton/mirror joint** se espelha a hierarquia selecionada, mantendo a conexão com a parte central do esqueleto.

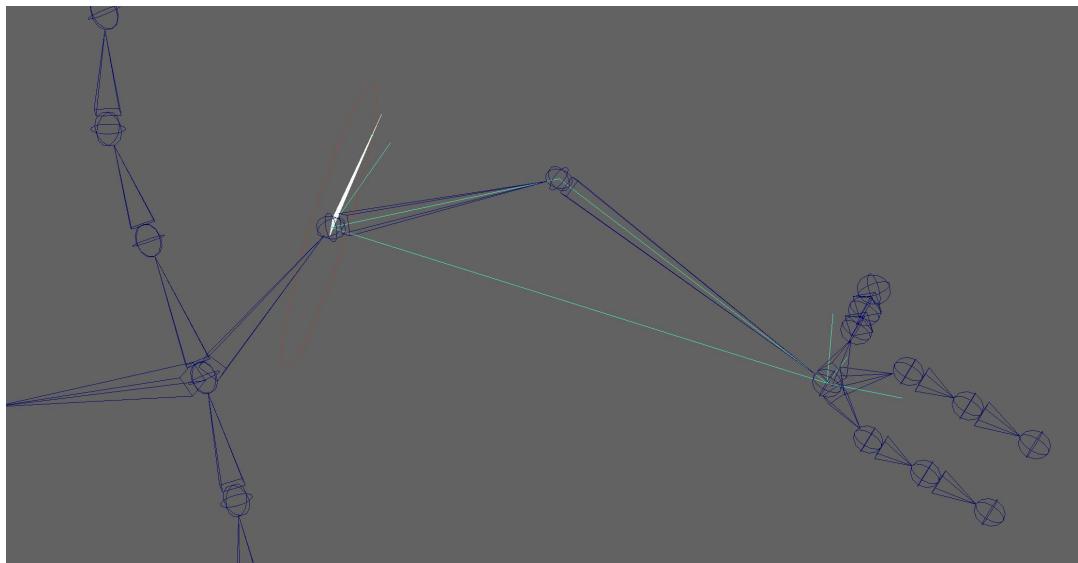


A opção **mirror across** define quais eixos de referência para o espelhamento, escolha o opção que mostre os dois eixos restantes ao eixo que deseja fazer a operação, é importante ter os joints nomeados nesta etapa já que os replicados ganham a nomenclatura de suas cópias, inclusive trocando partes do nome, como direito para esquerdo por exemplo, em **search for**, colocar o nome que deseja mudar e em **Replace with**, colocar a palavra que vai substituir a antiga.



c. IK Handle

A IK é um manipulador utilizado para conferir a um conjunto de *joints* uma movimentação semi-automatizada que simula articulações como braços. bastando com a ferramenta ativada (**keleton/Create IK Handle**) clicar no primeiro *joint* da hierarquia que será a base do ombro depois clicar no último *joint* que seria o pulso, agora quando movimentar a IK os ossos fazem movimento de rotação de forma semelhante a um braço para acompanhar o movimento.

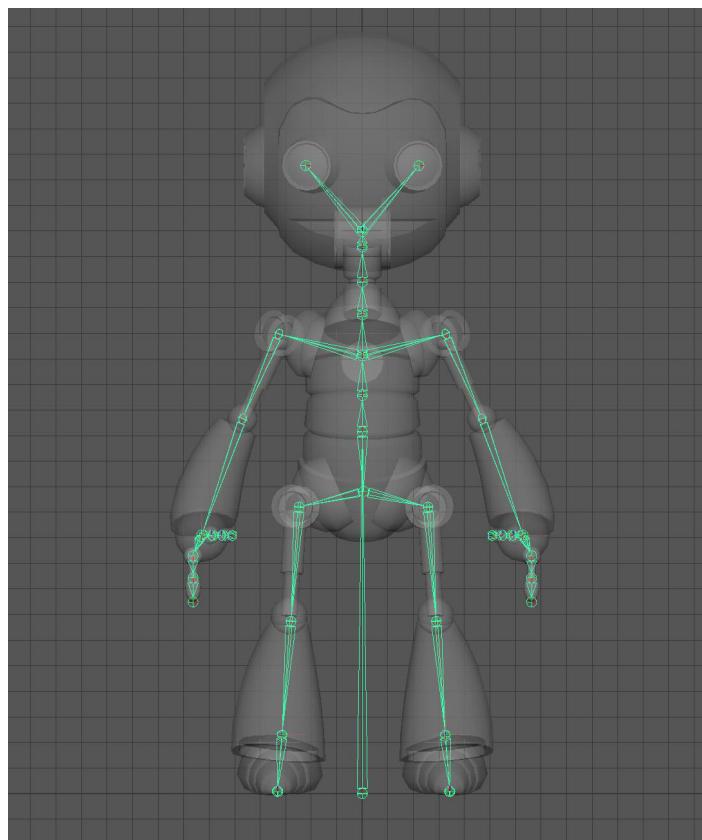


EXERCÍCIO IV

Vamos criar um *rigging* de um modelo feito anteriormente, nessa primeira etapa vamos criar o esqueleto do personagem com *Create Joint*, cada ponto de articulação deve ter seu *joint* de acordo com a estrutura do corpo, não esqueça de orientar espelhar os membros e nomear tudo antes de partir para próxima etapa.

Com este exercício aprendemos:

- » Criar e posicionar *joints*
- » Orientar *joints*
- » fazer espelhamento do esqueleto



02. Skin

Depois que o esqueleto está pronto, é hora unir os *bones* a mesh, usando o comando em **skin/bind**, a operação faz com que os vértices de uma determinada área passe a ser influenciados por um *bone* próximo, sofrendo as transformações deste.

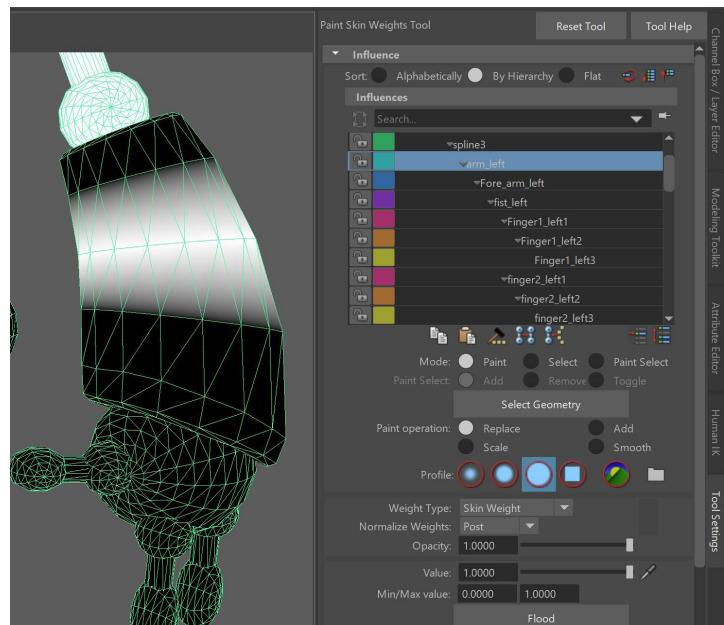
- » **Bind to:** Determina se a *mesh* vai se unir a uma hierarquia de *bones* ou somente os *joints* selecionados.
- » **Bind method:** Define como vão ser feitos os cálculos do *bind*, *Geodesic Voxel* costuma gerar bons resultados.
- » **Normalize weights:** Afeta a edição das influências da malha,
- » **Interactive:** quando é adicionado uma influência em um joint ela é retirada automaticamente nos outros, melhor para *rigging* de peças rígidas.
- » **Post:** É possível que vários *bones* afetem uma mesma parte da malha, adequado para personagens orgânicos mais complexos.
- » **Remove unused influences:** Remove os joints que ficaram sem influência do *bind*, normalmente é melhor desmarcar esta opção pois o cálculo pode remover um *joint* que precise ser utilizado.

As outras opções podem ser mantidas no Default, quase sempre é necessário fazer ajustes manuais.

a. Paint skin weights

Dificilmente o *bind* calcula todos os pontos de influência da malha corretamente, para editar estas influências basta acionar o comando **Skin/paint skin weights** habilita ferramenta de pintura de pesos destas influências, para visualizar a ferramenta como um todo é preciso acessar o **Tool settings**, outra forma de chegar a esta ferramenta é segurando o botão direito do mouse sobre um objeto com *bind skin*.

A *paint skin weights* é um pincel que pinta as influências dos ossos na malha, onde em cada um dos joints a influência máxima ou nula são representadas na malha respectivamente pelas cor preta, e com uma escala de cinza como influências intermediárias, no *tool settings* há uma lista com os *joints* que estão atrelados a esta malha, quando selecionados mostra pelo esquema de cores como aquele elemento específico está afetando a malha.



» **Painting operation:** dita como as influências vão se comportar no pincel, **Replace** troca a influência atual pela configurada no pincel, **Add** adiciona a influência do pincel a influência atual da malha, **Smooth** espalha a influência pela malha suavizando a pintura.

» **Profile:** formato de do pincel, pode fazer com que a influência do pincel seja rígida ou suavizada.

» **Normalize weights:** como dito no *bind skin*, em **post** mais de um *joint* podem afetar a mesma parte da malha e em **interactive**, quando uma influência é adicionada ela remove a influência dos outros *joints*. Atenção ao mudar esta opção a pintura de todo o modelo pode ser modificada, sendo o ideal fazer esta escolha na hora do *bind skin*.

» **Opacity:** adiciona gradativamente influencia a pintura, até chegar ao valor configurado no pincel.

» **Value:** valor de influência atual do pincel.

» **Min/max value:** Valor mínimo e máximo de influência do pincel.

» **Flood:** Aplica a influência do **Value** a toda a malha visível, usando a opção isolante, é afetar apenas a área isolada com a operação **Smooth**, suaviza a pintura por inteiro.

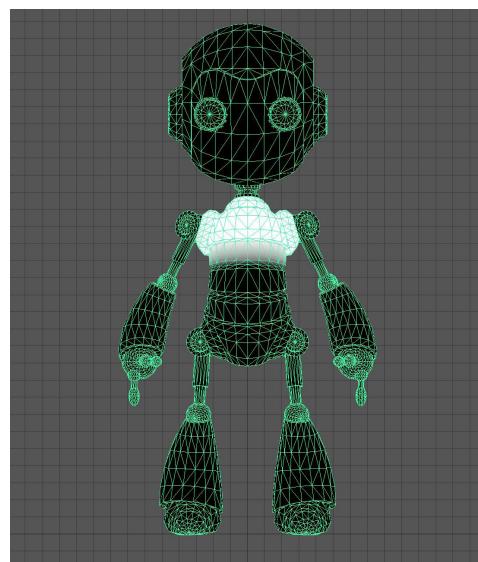
É interessante sempre testar a pintura movendo os joints a cada etapa, em modelos mais complexos, o polimento da pintura de pesos pode ser feita depois da etapa da criação de controladores.

EXERCÍCIO V

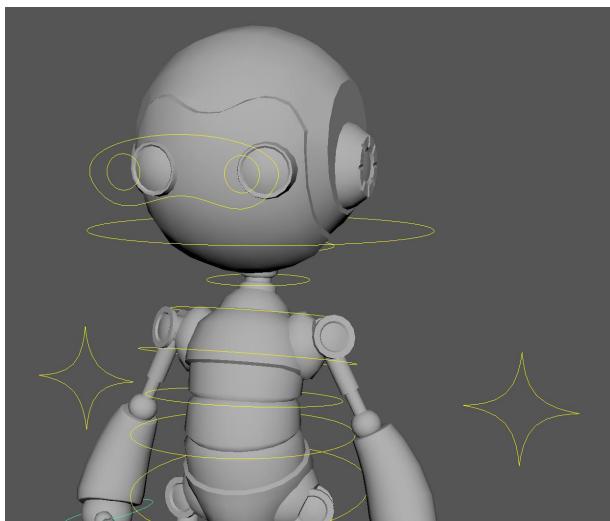
É hora de juntar a malha ao esqueleto através do *Bind Skin*. inicie uma pintura de pesos básica para o modelo que criou os *joints* anteriormente. não se esqueça de fazer pequenos testes para verificar o comportamento da malha, e sempre retorne o osso a posição inicial com undo (Z).

Com este exercício aprendemos:

- » Juntar esqueleto e malha
- » configurar a ferramenta de *Paint Skin Weights*
- » Definir a distribuição de influências na malha



03. Controladores

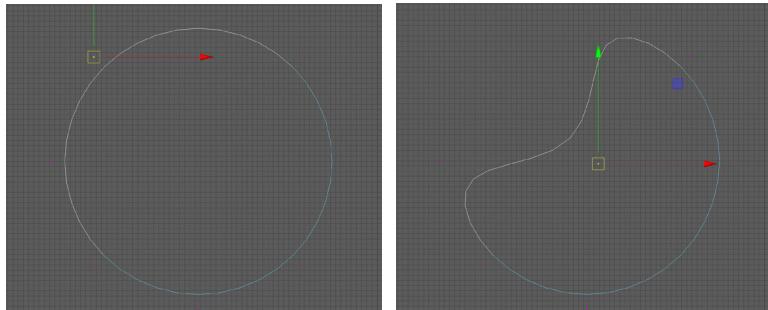


Normalmente não animamos os personagens diretamente nos ossos, costumamos utilizar manipuladores para esta tarefa, os controladores, que em sua maioria são curvas que vinculamos ao esqueleto para otimizar, organizar e facilitar a manipulação dos joints.

Encontramos as ferramentas para criar estas curvas na *Shelf Curves/surfaces* ou nos menus **Create/NURBS primitives** e **Create/Curve Tools**, caso precise fazer uma curva com formato diferenciado há a opção **EP curve tool**, onde pode-se criar formatos com as conexões feitas a cada clique, onde

Enter finaliza o trabalho e **Delete** volta um passo. Segurando com o botão direito do mouse sobre uma curva, se acessa a edição de seus componentes, em **control vertex** há a possibilidade de torcer a curva alterando sua aparência.

Antes de vincular as curvas ao restante do rigging é importante utilizar **freeze transformations**, as **nomear** de forma lógica e **deletar histórico** evitando eventuais problemas no funcionamento e mantendo a cena organizada.



Control vertex

04. Constrain

Constrains são formas de vincular movimentos de um objeto a outro, semelhante ao parentesco, porém permite relações mais específicas, seu uso mais comum é vincular as curvas aos joints do esqueleto, geralmente marcam os atributos que estão influenciando com a cor azul no *channel box*, e podem ser visualizado no *Outliner* e no *Hypergraph hierarchy*. Ao se fazer um constrain, o primeiro elemento selecionado vai controlar o segundo.

Umas das vantagens de utilizar constrains é a possibilidade de um objetos poder ser controlado com mais de um controlador, mas em atributos diferentes, ou mesmo que um único controlador afete dois elementos de forma diferente como por exemplo um controlador que com seu movimento controla a IK e com sua rotação controla o osso do pulso.

Os tipos de constrain são:

- » **Partent:** Vincula os atributos *translate* e *rotate*, sendo o constrain mais utilizado no rigging clássico.
- » **Orient:** Afeta unicamente o atributo *rotate*.

» **Point:** O atributo *translate* é afetado.

» **Scale:** Mais utilizado em *riggins* com transformações estilizadas, como *cartoon* por exemplo, como o nome diz vincula o atributo *scale*.

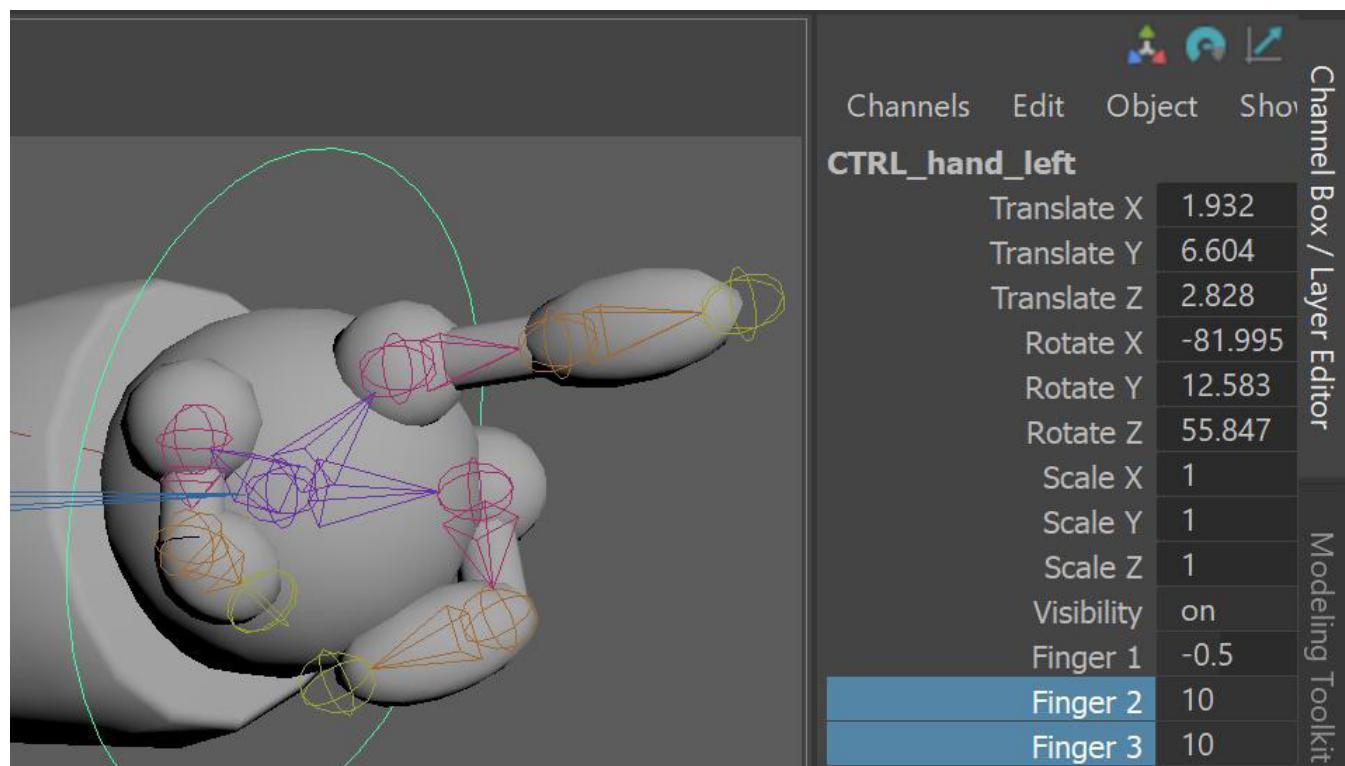
» **Aim:** Faz com que a rotação do objeto controlado sempre aponte para o controlador, seu uso mais comum é para fazer olhos de personagens.

» **Pole vector:** Um constrai específico para controlar IKs, ele controla para onde o centro da hierarquia da IK está apontando, o exemplo mais comum é o controlador de cotovelos e joelhos.

05. Set Driven Key

Alguns movimentos complexos porém repetitivos, como o abrir e fechar dos dedos de uma mão por exemplo, são complicados de se executar manualmente toda vez que são necessários, para automatizar este tipo de movimento costumamos utilizar o Driven key, este recurso vincula o atributo de um objeto as transformações de outros. Para acessar esta janela é preciso ativar o modo animation das ferramentas e acessar o caminho **Key/Set Driven key/Set**.

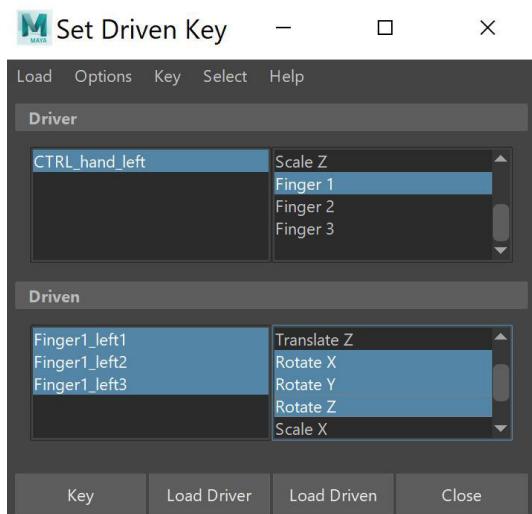
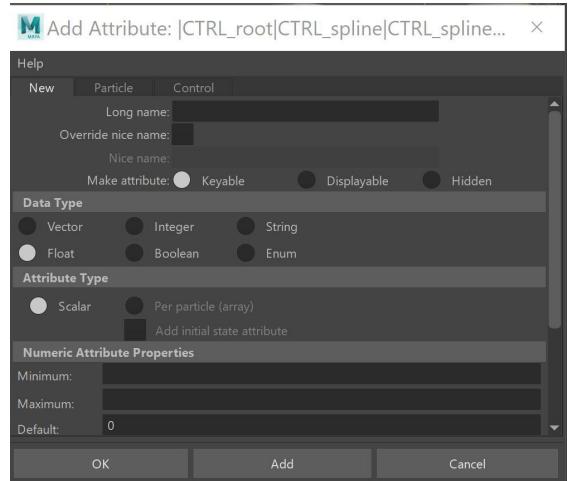
A primeira etapa na criação de um *drive key* normalmente é a criação de um atributo para o movimento desejado, no exemplo dos dedos o abrir e fechar específico de cada dedo possibilita várias possibilidades de posições para a mão, no **Channel Box**, **Edit/Add Attribute** com o objeto desejado selecionado, abre a janela de criação de atributos.



» **Long name:** nome do novo atributo.

» **Data Type:** define como vai ser efetuado o controle do atributo, **Float** que é uma graduação de um número até outro e **Boolean**, que equivale a ligado ou desligado, são os usos mais comuns.

» **Numeric Attribute Properties:** define o valor mínimo, máximo e padrão do atributo, no caso do dedo seriam respectivamente, dedo com a mão espalmada, com a mão fechada e com ela relaxada.



Depois que o atributo novo foi criado, na janela do Set Driven Key, se define o controlador em Load Driver, e os elementos a serem controlados em Load Driven, posteriormente os atributos a serem vinculados são selecionados e utilizando Key é possível ligar a transformação dos elementos.

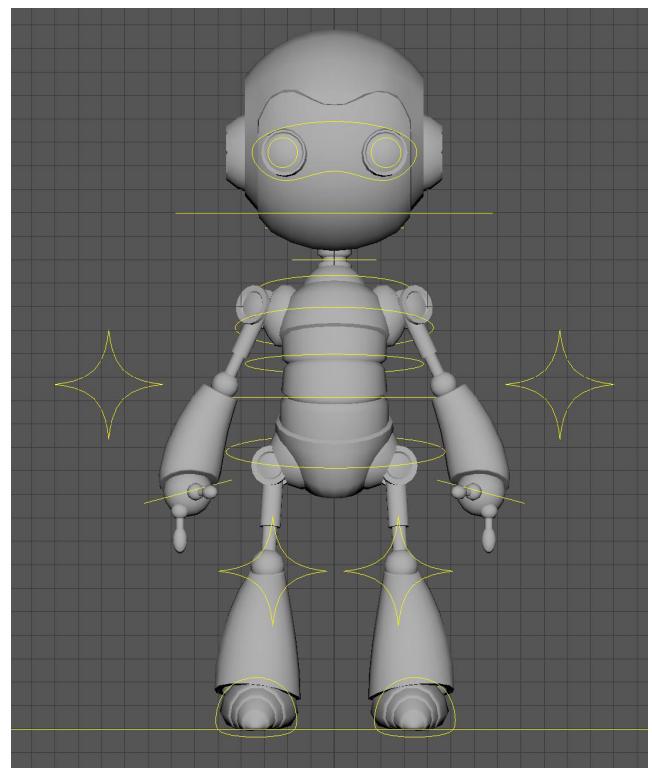
No exemplo dos dedos se utiliza o Key nas três posições básicas, posição relaxada e atributo na configuração default, os dedos fechados e o atributo no valor máximo e a mão espalmada com o atributo no valor mínimo.

EXERCÍCIO VI

Vamos finalizar o *rigging* de nosso personagem fazendo seus controladores, lembre-se de manter os controladores organizados e aproveite para refinar a pintura de pesos.

Com este exercício aprendemos:

- » Criar e anexar controladores
- » Configurar Set Driven Keys



PRINCÍPIOS DE ANIMAÇÃO

Compreender as ferramentas e técnicas aplicadas aos softwares que utilizamos para animar é importante, no entanto para o animador compreender conceitos que fundamentam a representação de movimentos também ajuda a obter bons resultados nos seus trabalhos. Existem algumas regras que nos ajudam a tornar nossas animações mais convincentes e carismáticas. Estes princípios estão no livro *The Illusion of Life: Disney Animation*, uma compilação das técnicas de animação utilizadas pelos Estúdios Walt Disney na década de 1930 e continuam orientando animadores até os dias atuais.

» Slow In and Slow Out

Para todo movimento há uma aceleração e uma desaceleração. No início, fim ou mudança de direção, costumam ter mais frames, a menos que seja um movimento muito drástico. No 3D boa parte deste trabalho é ajustado nas curvas do Graph Editor.

» Squash and Stretch

Ou esticar e achata, observa que qualquer ser vivo, ao se movimentar, apresenta modificações consideráveis em sua forma. Este efeito costuma ser usado para passar ideia de velocidade e maleabilidade. No maya dependendo do elemento isto pode ser feito com escala.

» Follow Through

Basicamente é a inércia, objetos presos a um corpo, como capas e acessórios, tendem a começar a se mover um pouco depois que o corpo principal e depois que o mesmo para os objetos presos tentam continuar se movendo.

» Anticipation

Um movimento antes da ação principal, como o ato de jogar o braço para trás antes de dar um soco ou dobrar os joelhos antes de pular, prepara o espectador para a ação. Quando a animação é para jogos este princípio tende a ser fraco nos movimentos dos jogadores para não passar impressão de falta de resposta aos comandos e exagerada nos inimigos para dar tempo de reação.

» Arcs

Exceto por algumas poucas criaturas, quase todo ser vivo segue uma linha levemente circular ao executar seus movimentos. Movimentar os elementos em arcos deixa a animação mais agradável e realista.

» Exaggeration

O exagero nas posições e movimentos torna uma ação ou expressão mais convincente, ou até mesmo engraçada. Evite movimentos contidos para ações enérgicas.

» Secondary Action

Criar ações secundárias que dão suporte a ação principal ajudam as animações mais carismáticas e podem ditar o humor da ação, como um personagem que tem como ação principal andar, mas pode assoviar ou resmungar enquanto anda.

» Staging

Ou atuação, tem ligação com posicionamento e pose durante uma animação. o animador deve atuar de forma semelhante a um diretor de cinema se preocupando com enquadramentos e fotografia.

» Solid Drawing

Posições que reforçam o efeito 3D, evitem poses muito simétricas e trabalhar sobreposições que reforçam a ideia de camadas. No maya sempre anime os personagens nos três eixos, mesmo que de forma suave.

» Appeal

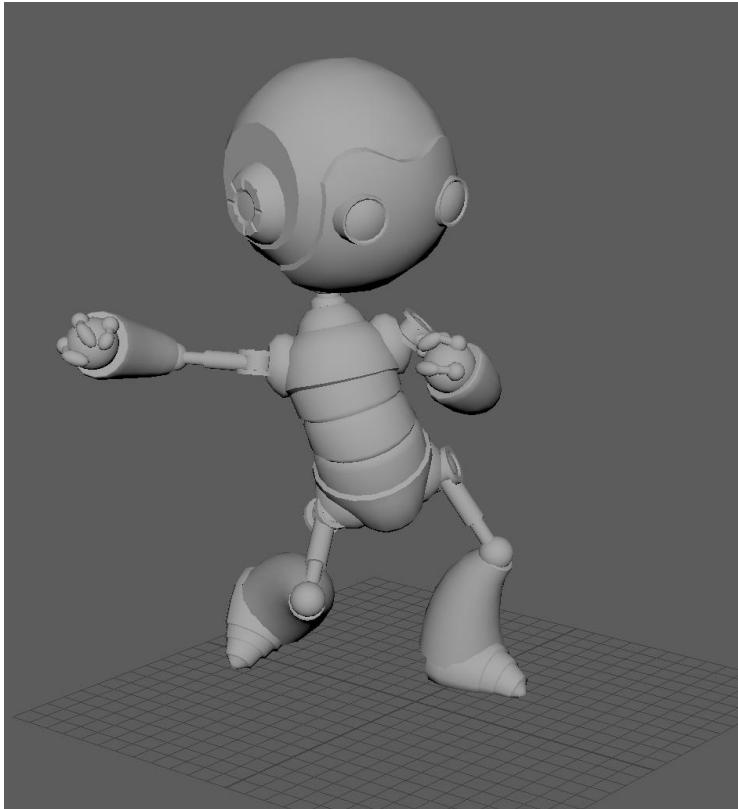
Se traduz em carisma nos movimentos, é tentar imprimir personalidade nos movimentos, como por exemplo diferentes formas de andar, zangado, gracioso ou desengonçado.

EXERCÍCIO VII

Faça uma animação utilizando o personagem que teve o rigging feito anteriormente, procure aplicar o máximo do que vimos neste módulo, como polimento de curvas e tente explorar os princípios de animação.

Com este exercício:

- » exercitamos animação
- » colocamos em prática o que vimos durante o módulo





TEXTURIZAÇÃO

INTRODUÇÃO

INTERFACE E COMANDOS BÁSICOS

ATRIBUTOS BÁSICOS DE MATERIAIS

TOOLS

CAMADAS

INICIANDO O PROJETO

BAKE MESH MAPS

MÁSCARAS

SMART MATERIALS

ADD EFFECTS

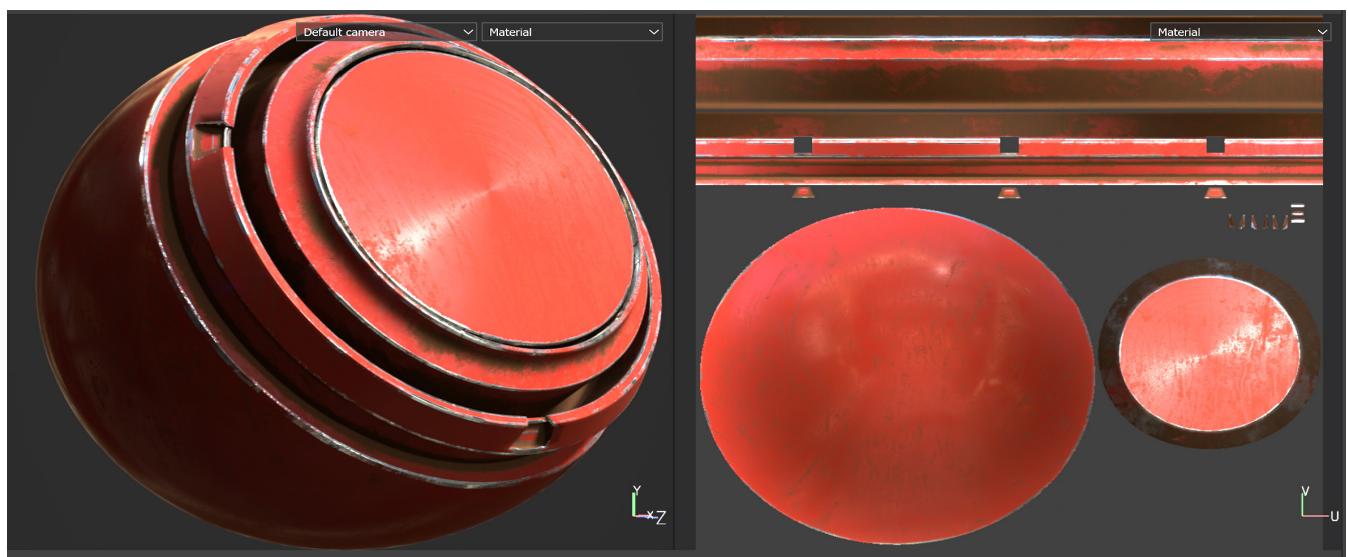
EXPORTAÇÃO DE TEXTURAS

INTRODUÇÃO

O processo de texturização de um objeto 3D é capaz de transformar profundamente um modelo, atributos como cor, brilho, flexibilidade, entre outros, definem elementos importantes como qual o seu tipo de material, seu estado de conservação e é capaz de os tornar mais carismáticos e expressivos. Um mesmo personagem pode mudar completamente de aparência com texturas diferentes, algumas *Skins* podem ser criadas apenas alterando as características dos materiais, mudando a intenção de um personagem, sombrio, futurista, angelical, são diversas possibilidades, obviamente que a modelagem tem igual peso em estas questões, mas a texturização tem o poder de fazer isso de forma prática e econômica.

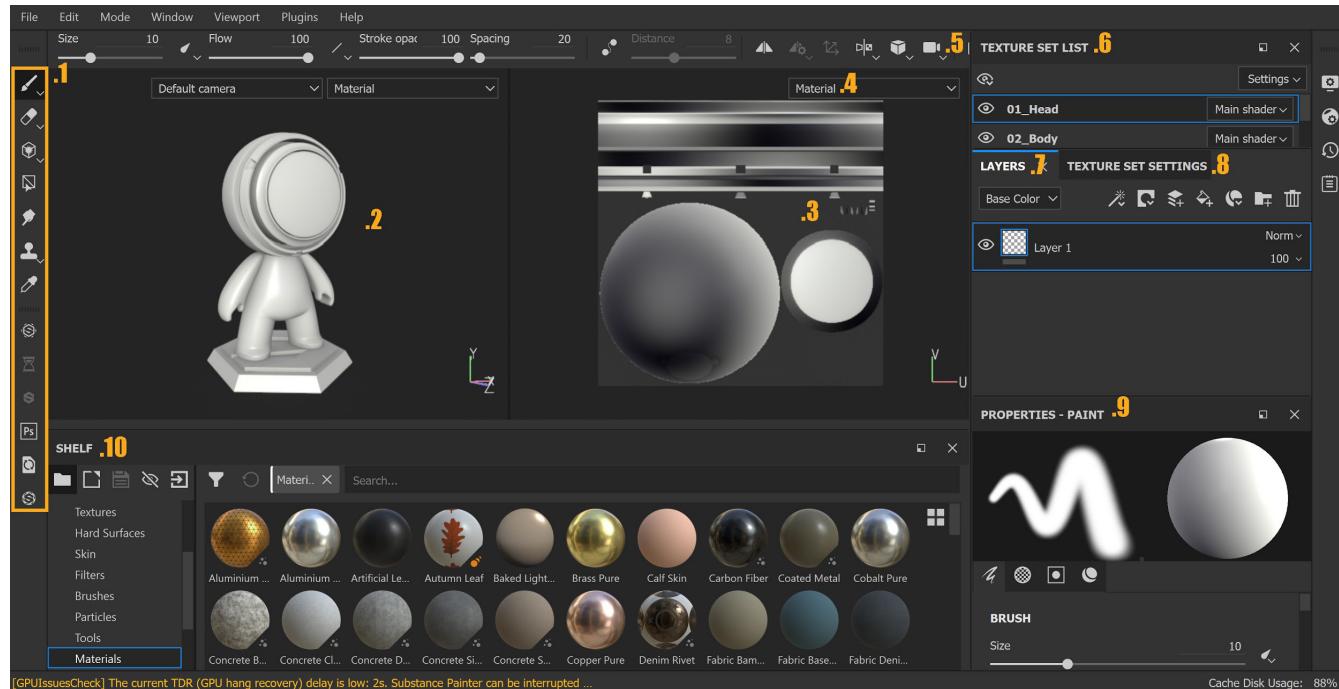
No módulo anterior os materiais foram explicados de forma básica, eles são o que nos permitem enxergar a superfície criada a partir da união de vertex e edges que compõem o *wireframe* de nossos objetos 3D, controlando a forma como visualizamos o modelo, o que acontece muitas vezes através de texturas projetados no mapa UV, estas imagens permitem levar estas características a diversas plataformas diferentes.

A principal ferramenta utilizada neste módulo será o Substance Painter, extremamente intuitiva e poderosa, traz diversos recursos pré configurados e tem opção de aplicar a texturização diretamente sobre o objeto 3D, é uma ferramenta fácil de aprender e entrega ótimos resultados.



INTERFACE E COMANDOS BÁSICOS

A interface do *Substance painter* é intuitiva e compacta, apesar da quantidade de recursos disponíveis passar a ideia de complexidade, são ferramentas que pertencem a menus específicos, quando comparamos suas funções o software se mostra de fácil utilização.



- » **Tools:** ferramentas que permitem editar manualmente a texturização.
- » **3D view:** visão do modelo 3D.
- » **2D view:** Visão do mapa UV do modelo 3D.
- » Visualiza qual parte do material está sendo visualizado.
- » Opções que lidam com a visualização do modelo.
- » **Texture set list:** mostra os materiais presentes no objeto base.
- » **Layers:** camadas de pintura do material,
- » **Texture set settings:** configurações do material que será gerado.
- » **Properties:** Propriedades da ferramenta selecionada no momento.
- » **Shelf:** coletânea de recursos disponíveis para utilização.

O Substance Painter tem cenas base que mostram possibilidades de efeitos em **File/ Open sample**, entre eles o arquivo **MeetMat**, um modelo neutro, bom para testar as primeiras ferramentas.

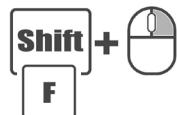
01. visualização 3D/2D

A área de trabalho do Substance Painter segue um padrão semelhante a de outros softwares de modelagem 3D, sendo bem próxima a do próprio Maya, mas ao invés das viewports ortográficas temos uma janela para trabalharmos diretamente sobre o mapa UV, podemos alternar as visualizações com a tecla **F1**, **F2** mostra apenas a vista 3D e **F3** a visão 2D.

» Girar a câmera:



» Centraliza a camera



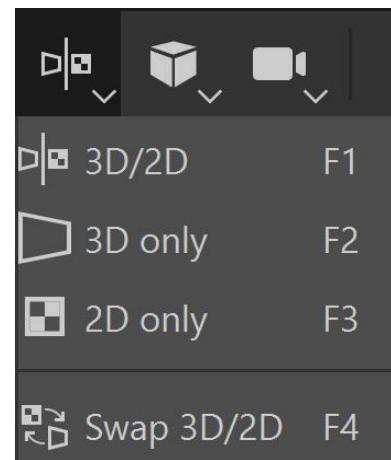
» Zoom:



» Move a câmera:



» **move o foco de iluminação**



Caso queira aumentar sua área de trabalho, a tecla **TAB** esconde ou mostra a interface. Caso sua interface esteja desconfigurada, o comando **windows/ Reset UI** restaura o *layout* padrão do software.

ATRIBUTOS BÁSICOS DE MATERIAIS

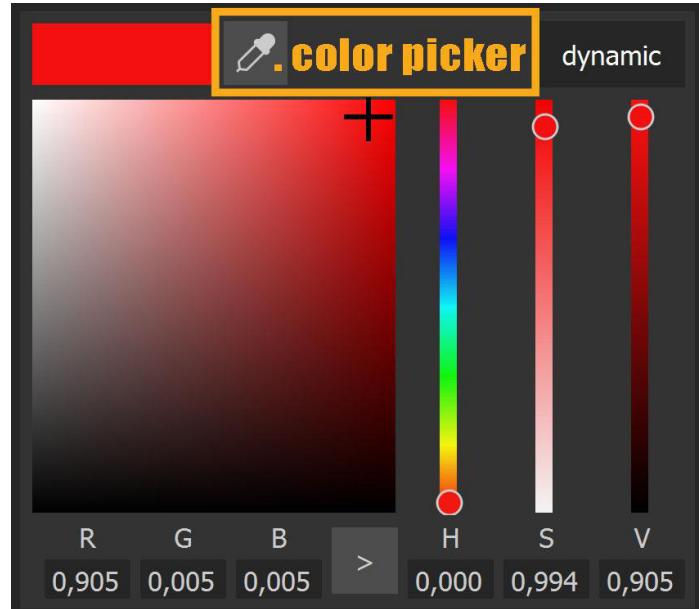
Antes de explicar as ferramentas é importante entender as características que podemos alterar no material. Saber configurar estes parâmetros é muito importante, eles definem pontos importantes do seu trabalho, trabalhá-los corretamente pode diferenciar vidro de madeira, algo velho e desgastado de algo novo e brilhante, conseguir detalhes de volume sem alterar a quantidade de malha do modelo entre outros.



01. Color

Controla a cor sólida do objeto independente de efeitos de luz o tipo do material, trabalho no padrão RGB (Red, Green, Blue), e tem os seguintes atributos:

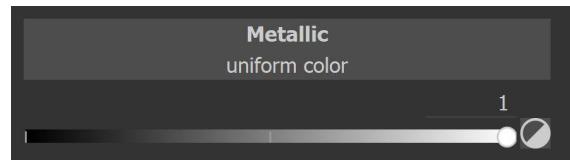
- » **Hue:** Que determina a matiz, ou seja, qual cor de fato está sendo trabalhada.
- » **Saturation:** Vividez da cor, quanto mais alta a saturação, mais viva e colorida e quanto menor a cor se torna menos vibrante tendendo ao cinza.
- » **Brightness:** Brilho ou luminosidade da cor, no valor mediano mostra a cor pura, quando alto deixa a cor mais leve tendendo ao branco e quando baixa deixa a cor mais sólida tendendo ao preto.



Color Picker, é uma ferramenta capaz de selecionar uma determinada cor para sua paleta, está com pode estar em qualquer local de tela inclusive fora da janela do Substance.

02. Metallic

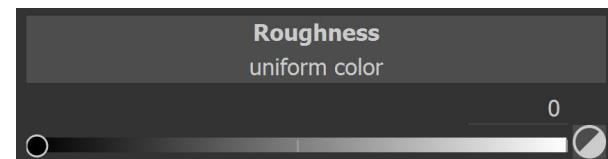
Determina o quanto o objeto é metálico, diz respeito a intensidade do brilho e cor do material, trabalha com uma graduação do preto para o branco onde totalmente preto é completamente sem o efeito e branco totalmente metálico.



Trabalhando áreas com diferentes quantidades de metallic, é possível ressaltar arranhões e desgaste.

03. Roughness

Este atributo define o quanto a superfície do material reflexiva, falando em termos físicos é a diferença entre o liso e áspero, quanto mais alto o *Roughness* mais liso e com consequência mais reflexivo é o material.



Este parâmetro também trabalha com escala de preto para branco, quando menor o Roughness mais reflexiva é a área do material.

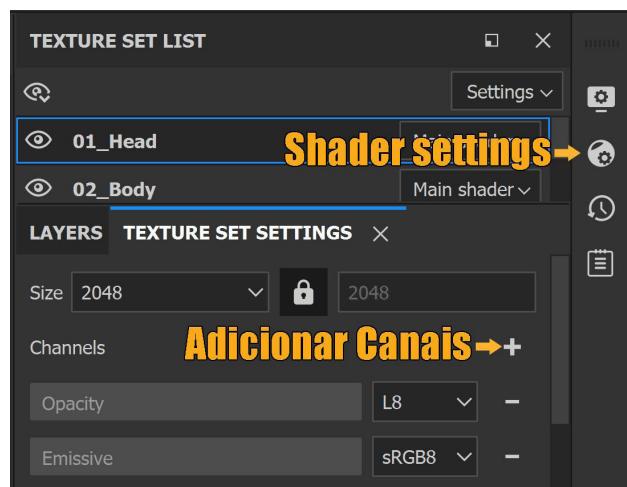
04. Height/Normal

Estes atributos simulam volume no modelo alterando a forma com que a luz se comporta sobre a superfície, normalmente no Substance o normal map é ligado a transformações mais automatizadas como *hard surfaces* ou gerados a partir de modelos 3D. Já o Height, trabalha com uma escala de -1 a 1 para determinar alto e baixo relevo.



Este tipo de volume é ideal para detalhes finos que não se afastem muito da superfície, evitando que criar estes volumes com malha 3D, o que seria bem mais pesado.

Existem outros dois atributos importantes que não vem habilitados, **Opacity** e **Emissive**, para ativá-los é preciso adicionar estes canais ao material no **Texture set Settings**, no sinal de “+” ao lado de **channels**.

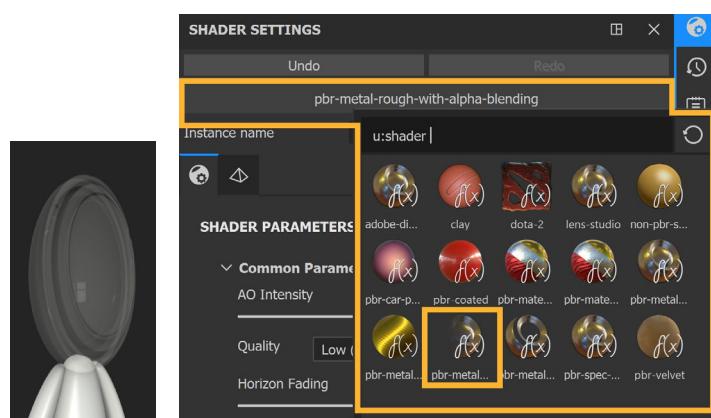


Entre várias coisas o **Texture set settings**, pode visualizar e configurar o tamanho dos mapas do material selecionados.

O **shader settings** tem configurações mais profundas e tem outros shaders pré-configurados.

05. Opacity

Controla a transparência do material, também trabalha com uma escala de preto, transparente, a branco, opaco, variando o valor de 0 a 1.

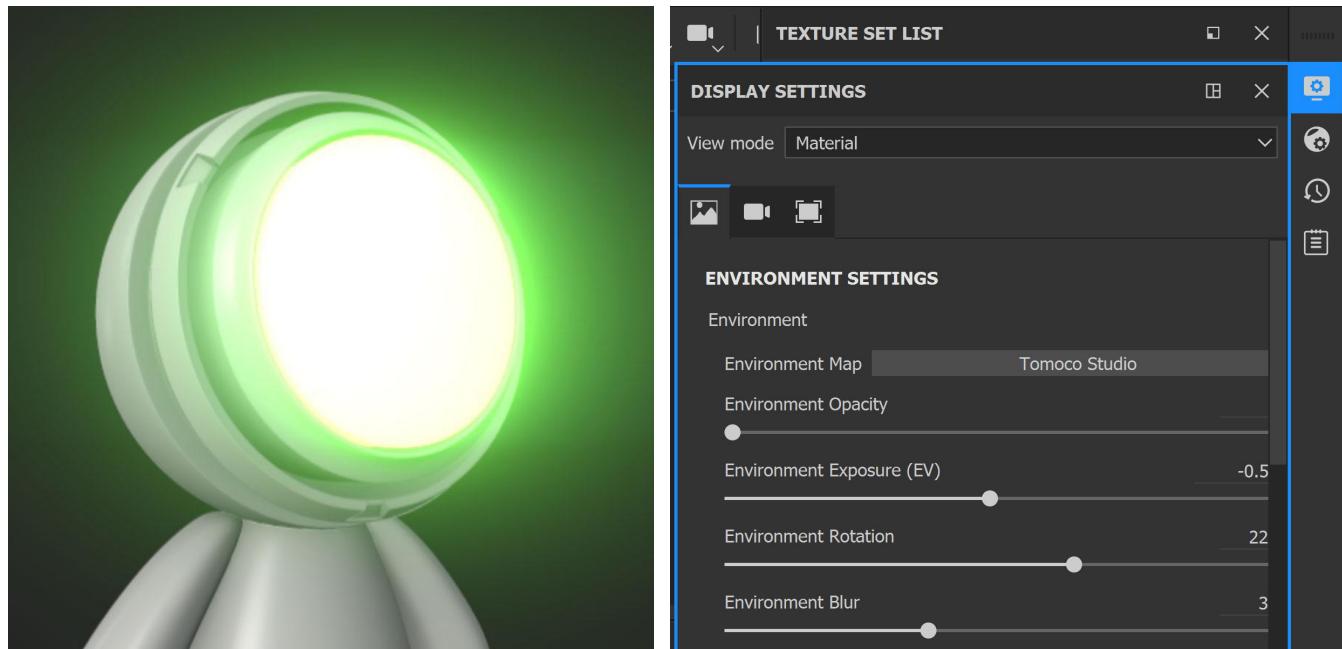


O shader padrão não aceita **opacity**, no primeiro botão do **shader settings** existem outros shaders, para habilitar transparência, é preciso além de adicionar o canal, mudar o shader para **pbr-metal-rough-with-alpha-blending**.

06. Emissive

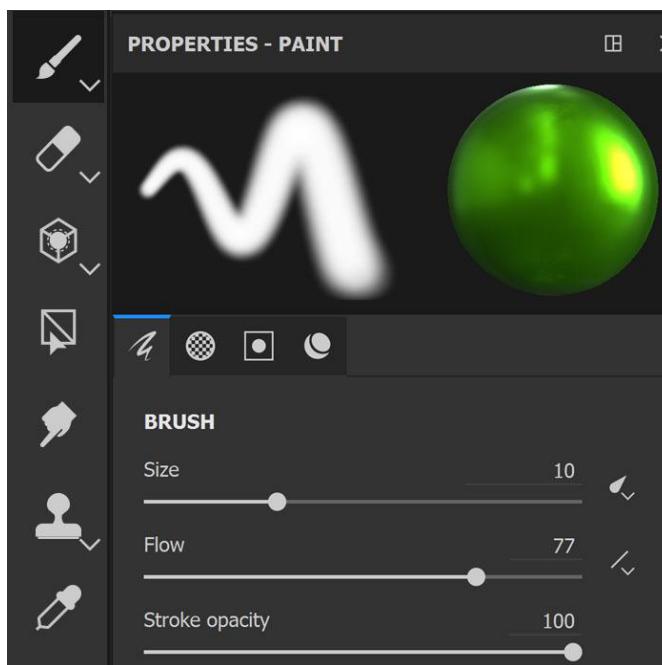
Adiciona luz ao material, como vários atributos, utiliza claro e escuro para determinar a intensidade, mas pode utilizar cores para determinar a coloração desta luminosidade.

Ao adicionar o canal emissive, é preciso fazer algumas configurações para que ele possa ser visualizado, em *Shader settings* existe a opção **Emissive Intensity**, e em **Environment Settings** aumente o **Environment Exposure** e por último ative os **Post Effects** e a opção **Glare**.



TOOLS

Em nossa barra de ferramentas temos várias opções de edição manual de textura, ao selecionar uma delas, suas configurações ficam disponíveis no menu **Properties**, que além das características próprias de cada ferramenta tem uma pré-visualização de seus resultados.



01. Paint



Pincel que pode pintar na textura vários os canais, *color*, *metallic*, *roughness*, *height*, *opacity* e *emissive*, sendo uma ferramenta útil para modificações pontuais, os colchetes ("[" e "]") respectivamente aumentam e diminuem o tamanho do *Brush* e a tecla **Shift** permite fazer traçados em linha reta.



PROPERTIES

- » **Size:** Tamanho do brush.
- » **Flow:** Fluxo de pixels, equivale a diminuir a quantidade de tinta do pincel.
- » **Stroke opacity:** Transparência dos elementos adicionados.
- » **Spacing:** Espaço entre um *Alpha* (pincelada) e o próximo.
- » **Angle:** Ângulo do *Alpha*, funciona com alphas de formato irregular.
- » **Follow path:** Faz o alpha seguir o caminho do traçado.

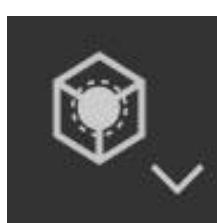
- » **Jitter:** As configurações de jitter acrescentam aleatoriedade a size, flow, angle e position. São ótimos para deixar a pintura orgânica.
- » **Alpha:** Formato do brush, há uma galeria de formatos disponíveis.
- » **Hardness:** suavização das bordas do pincel, quanto maior o valor mais nítido e quanto menor mais borrado.

02. Eraser

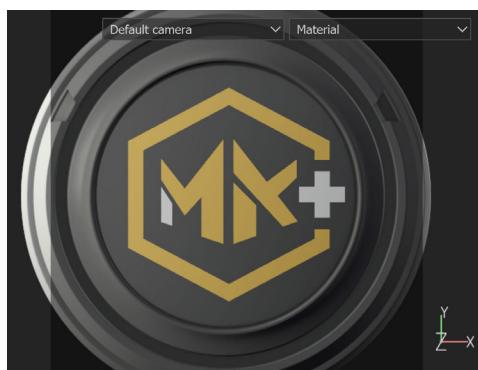


A borracha funciona de forma muito semelhante à ferramenta *Paint*, tendo as mesmas configurações, com a diferença que ela remove elementos da textura ao invés de adicionar.

03. Projection



Projeta uma imagem sobre o modelo 3D usando um brush, esta imagem é selecionada clicando em **Color Base**, para posicionar a imagem que será projetada mantenha pressionada a tecla **S**, sua manipulação é semelhante a movimentação da câmera. para adicionar uma imagem de fora do Substance basta iem em Shelf/ Import resources, a imagem pode ser nomeada, e pode se escoller se ela vai ficar salva a sessão, este projeto ou a shelf geral.

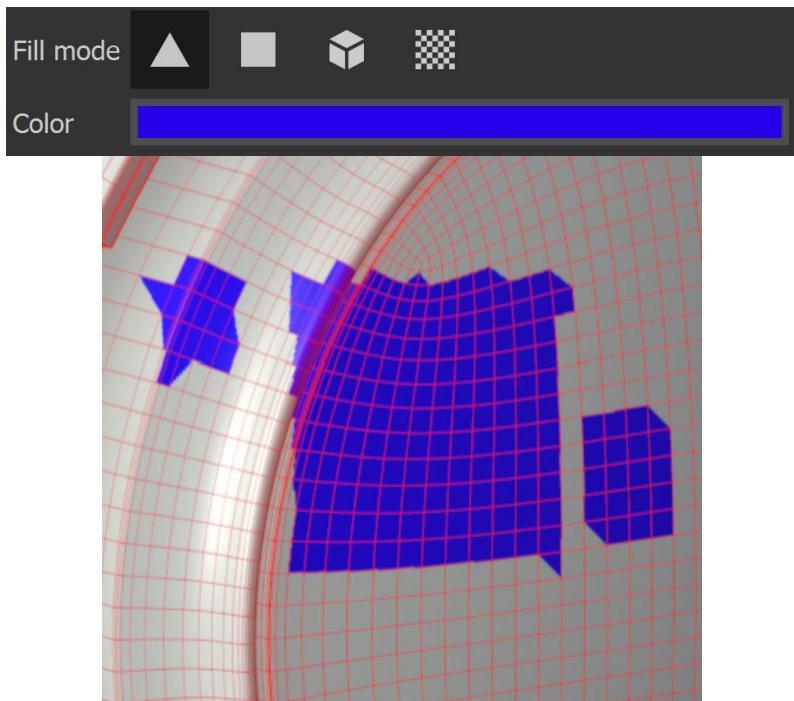


A projeção é feita de acordo com o posicionamento da câmera, pressionando Shift quando move a câmera ela vai travar em ângulos retos.

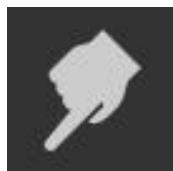
04. Polygon Fill



Esta ferramenta utiliza a malha 3D e o mapa UV para preencher áreas específicas do modelo, utiliza apenas o canal de cor, por esse motivo geralmente é utilizada para pintar máscaras (aportaremos máscaras em breve). Suas opções de preenchimento são triângulo, quadrado, objeto e UV shell.



05. Smudge



Tem funcionalidade bem próxima a do pincel, no entanto ao invés de adicionar ele somente movimenta os elementos que já existem criando um efeito de “borrado”.



CAMADAS

As Layers são um recurso muito útil não somente no Substance, mas em vários softwares de design. Estas camadas de trabalho permitem sobrepor elementos, mesclando ou ocultando os mesmos de acordo com a posição no menu, estando as camadas em ordem de prioridade de cima para baixo. Poder trabalhar elementos isoladamente em camadas evita retrabalho, já que se uma camada precisar ser removida ou alterada ela não danifica o restante do trabalho.

As layers são dispostas em um bloco vertical, onde as camadas superiores têm prioridade de visualização, para alterar a ordem das camadas basta arrastar a camada escolhida para a nova posição.



1. Ícone do olho no início da layer desabilita e habilita a visualização da camada.
2. Miniatura do mapa referente aos elementos presentes nesta camada.
3. Nome da camada pode ser alterado dando duplo clique sobre a fonte.
4. Modo de mesclagem da camada. Determina como os elementos da camada vão se misturar com os outros elementos. É possível determinar o modo de mesclagem de forma separada para cada canal na caixa no início do menu.
5. Opacidade da camada. Valor gradativo que vai de 0 a 100% tem efeito sobre a transparência.

OPÇÕES BÁSICAS DAS LAYERS:

» Add a layer:



Acrescenta uma nova Layer vazia.

» Remove Selected layer:



Deleta a camada selecionada.

» Add a folder:



Acrescenta uma nova pasta.
Importante para organizar o projeto.

» Add a fill layer:



Adiciona uma layer preenchida.
Muito útil para recobrir superfícies.

Uma das vantagens de uma **Fill Layer**, é que seus elementos continuam editáveis, ao contrário acontece quando se utiliza o pincel, sendo estes parâmetros acessados no menu **Properties**.

Manter a organização das camadas ajuda no fluxo do trabalho, um projeto pode ter dezenas de camadas. Nomear e agrupar em pastas organizadas ajuda a encontrar mais rapidamente os elementos e também ao precisar mudar a ordem ou desativar a visualização de partes específicas da texturização.

EXERCÍCIO I

Use os samples disponíveis no *substance* para fazer uma pintura estilo *Toy Art*, procure trabalhar ao máximo as ferramentas mostradas até o momento.



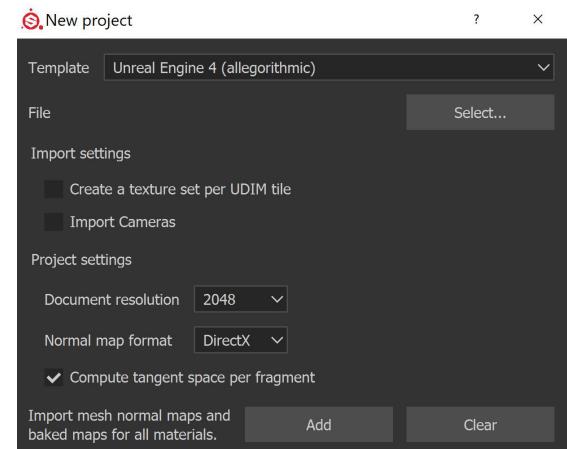
Nesse exercício aprendemos:

- » Manipular a interface básica
- » Organizar camadas e pastas
- » Explorar os diferentes canais na textura
- » Utilizar e configurar nossas ferramentas

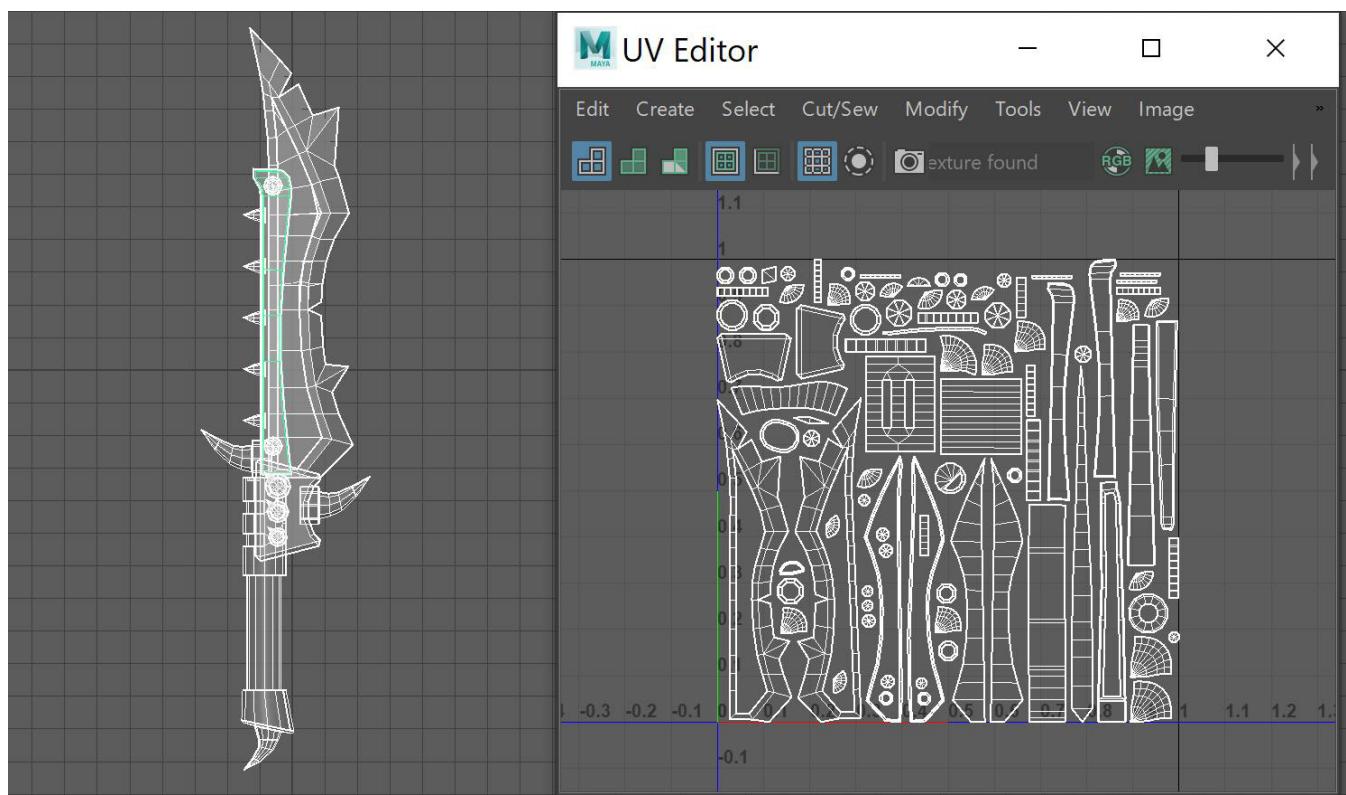
INICIANDO PROJETO

O *substance painter* não cria modelos 3D, ele tem a função de criar textura e materiais, sendo necessário criar os objetos em outro software e incorporá-los em um novo projeto dentro do substance, **File/new (Ctrl+N)**, é importante que os modelos estejam com mapa UV criado.

- » **Template:** Configurações básicas dos *shaders*, deve ser escolhido de acordo com a software que vai renderizar o produto final, estando disponíveis as principais *engines* de jogos, em nosso curso usaremos. **Unity**.
- » **File:** Botão onde anexamos os arquivos que vamos texturizar, os formatos mais utilizados são **.FBX** ou **.OBJ**.
- » **Document Resolution:** Tamanho dos mapas gerados, define a qualidade de resolução da textura, mas aumenta o custo em processamento.
- » **Normal map format:** Define como vai ser calculado o mapa de normais, sendo necessário saber qual opção correta para a engine que está utilizando. O **Unity** usa **OpenGL**.



Antes de importar o arquivo no *substance*, certifique se de exportar corretamente o modelo. Revise se o mapa UV está adequado, nomeie as *meshes* e os materiais, aplique o *freeze transformations* e delete o histórico, isto deixa o arquivo mais leve, organizado e diminui a chance de eventuais bugs de visualização.

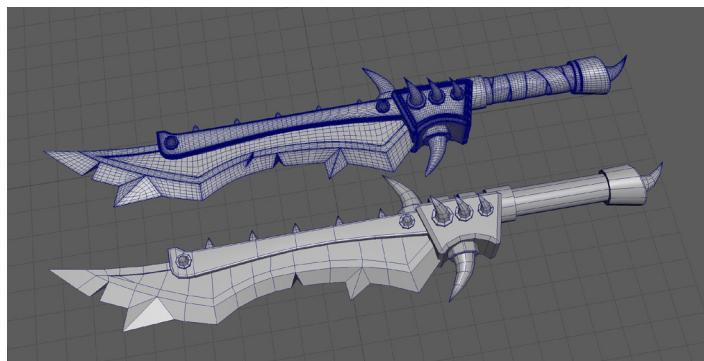


BAKE MESH MAPS

Uma das vantagens do **Substance** são os cálculos feitos nas malhas em que trabalha, detalhes como saber onde existem cantos, para aplicar efeitos de desgaste ou acúmulo de sujeira por exemplo, há diversos smart materials, que trazem ótimos resultados pré configurados para diversas situações, mas para que isso ocorra é necessário preparar o modelo para tanto, a opção **Texture settings/Bake mesh maps** executa os cálculos necessários de acordo com as configurações desejadas.

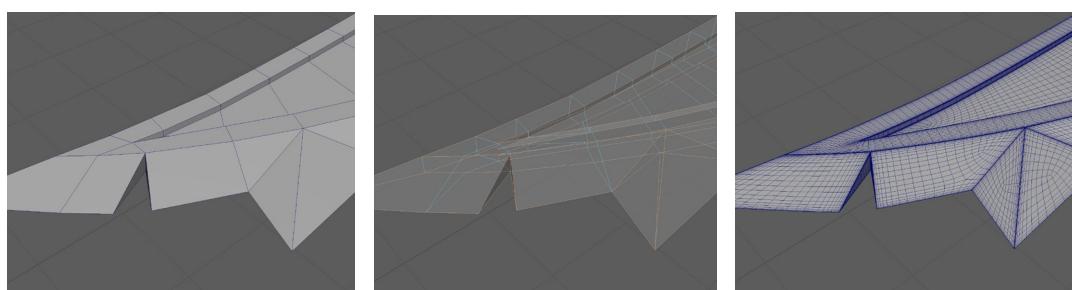
01. High definitions meshes

Aqui é possível anexar uma malha *high poly* do modelo que pretende trabalhar, é ótimo para suavizar a malha e acrescentar detalhes de normal, transportando detalhes finos da mesh *high* para a *low*. Caso não tenha uma malha com alta definição, pode-se usar a própria malha atual para fazer os cálculos marcando a caixa **use low poly mesh as high poly mesh**.



Utilizando o Autodesk Maya para criar a versão *High Poly* é interessante usar a ferramenta **Smooth** para suavizar a malha. Nesta etapa é possível acrescentar detalhes de volumes como arranhões, altos e baixos relevos.

Para usar o Smooth suavizando a malha, mas mantendo as quinas rígidas, é preciso criar *loopings* de sustentação. Onde há cantos afiados é necessário colocar várias edges juntas para quando a suavização acontecer estas áreas se mantenham inalteradas, as ferramentas *bevel* e *extrude* são adequadas para este propósito.



Um detalhe importante a ser ressaltado é que a high mesh não precisa ter um mapa UV aberto corretamente.

02. Dilatation with/max frontal distance/max rear distance

Estas opções dizem respeito à diferença de volume entre a *low* e a *high* mesh, é importante não exagerar na diferença de volume entre os dois modelos, mas estas opções ajudam a ajustar as distâncias caso necessário.

03. Antialiasing

Suaviza os mapas tirando o serrilhado, exige processamento do computador na hora do *bake*.

04. Match

Considera as partes do modelo juntas (**always**) ou separadas por nome (**by mesh name**) na hora de calcular os mapas fazendo as peças marcarem ou não as normais umas das outras, habilitando *by mesh name* o *substance* combina as partes procurando nome, identificando as meshes pelos **High poly mesh suffix** e **Low poly mesh suffix**.

05. ID

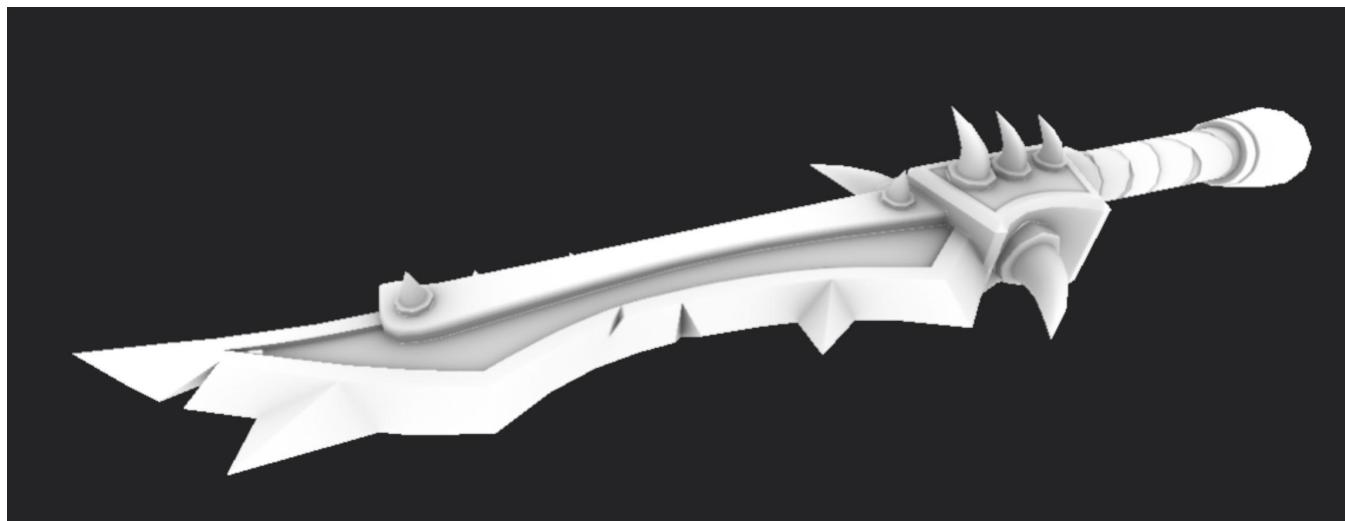
Cria seleções rápidas baseadas nas cores dos materiais da *mesh high poly*, marcando a opção **Material color** ou caso esteja usando o software zbrush existe a opção de usar mesh **ID/poly group**.

O ID é muito valioso para criar máscaras, separar diferentes materiais ou peças muito próximas.



06. Ambient occlusion

A oclusão é um efeito de sombra de contato, quando objetos se encostam eles costumam gerar sombras uns nos outros e este parâmetro gera um mapa que simula este efeito. **Secondary Rays** determina a intensidade das sombras e **Self Occlusion** se vai calcular as peças *Always* ou *by mesh name*.

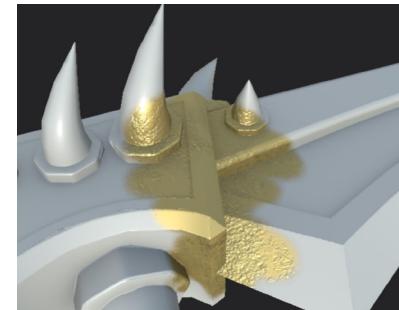
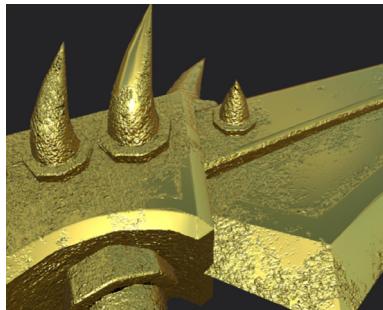


07. Curvature/Position/Thickness

Estes parâmetros calculam o formato e posição do objeto, ajudando a aplicar os efeitos especialmente nos *Smart materials*.

MÁSCARAS

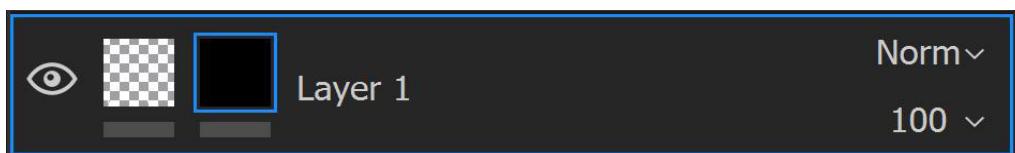
As máscaras são subcamadas que ocultam ou revelam elementos que estão presentes nas camadas em que estão vinculadas, são um recurso valioso ao facilitar o trabalho, já que com elas podemos aplicar a pintura em pontos específicos sem que a mesma saia da área delimitada. O funcionamento da máscara consiste em mapas em preto e branco, onde preto oculta os elementos e branco os revela e a escala de cinza funciona como um gradiente entre o visualizar ou não a camada.



01. Add mask



Cría máscaras dos tipos: *Add white mask*, *Add black mask*, *Add bitmap mask*, *Add mask with color selection*. Ela pode ser visualizada e configurada na miniatura nova logo ao lado da miniatura da camada. Mudando a seleção em azul se alterna entre o conteúdo da *layer* e o da máscara.



a. Add white mask/Add black mask

Adiciona uma máscara completamente branca onde se pinta os detalhes que se quer esconder, ou uma máscara completamente preta onde se pinta o que se quer revelar.

b. Add bitmap mask

Cria uma camada a partir de uma imagem em escala de cinza. Como todo mapa de textura deve ser um quadrado perfeito e afeta o objeto segundo seu mapa UV.

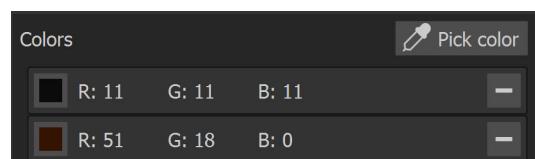


c. Mask with color selection

Utilizando o **Color ID** cria uma máscara a partir das cores selecionadas, é uma forma muito versátil de criar máscaras rápidas se a *high mesh* estiver bem preparada, já que se aproveita os volumes da malha com maior densidade para se aplicar materiais que nem sempre estão presentes na malha na *low poly*.



Adicione ou remova as cores desejadas no menu *Colors*.

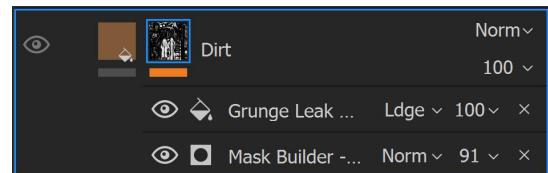


SMART MATERIALS

Smart materials são texturas pré-prontas que simulam diversos materiais como tecidos, madeiras, metais, e em diversos estados como pintados sujo ou corroído. Vêm organizados em pastas com diversas camadas, cada uma delas com vários parâmetros para serem configurados. Para anexar um *smart material*, basta selecionar na **shelf** e arrastar a miniatura para cima do modelo ou apertar o botão **Add a smart material** no menu de camadas.



É importante notar que cada camada e cada máscara tem seus próprios parâmetros que variam muito de acordo com o *smart material* trabalhado, sendo comum a princípio prestar atenção exatamente em que área se está trabalhando.

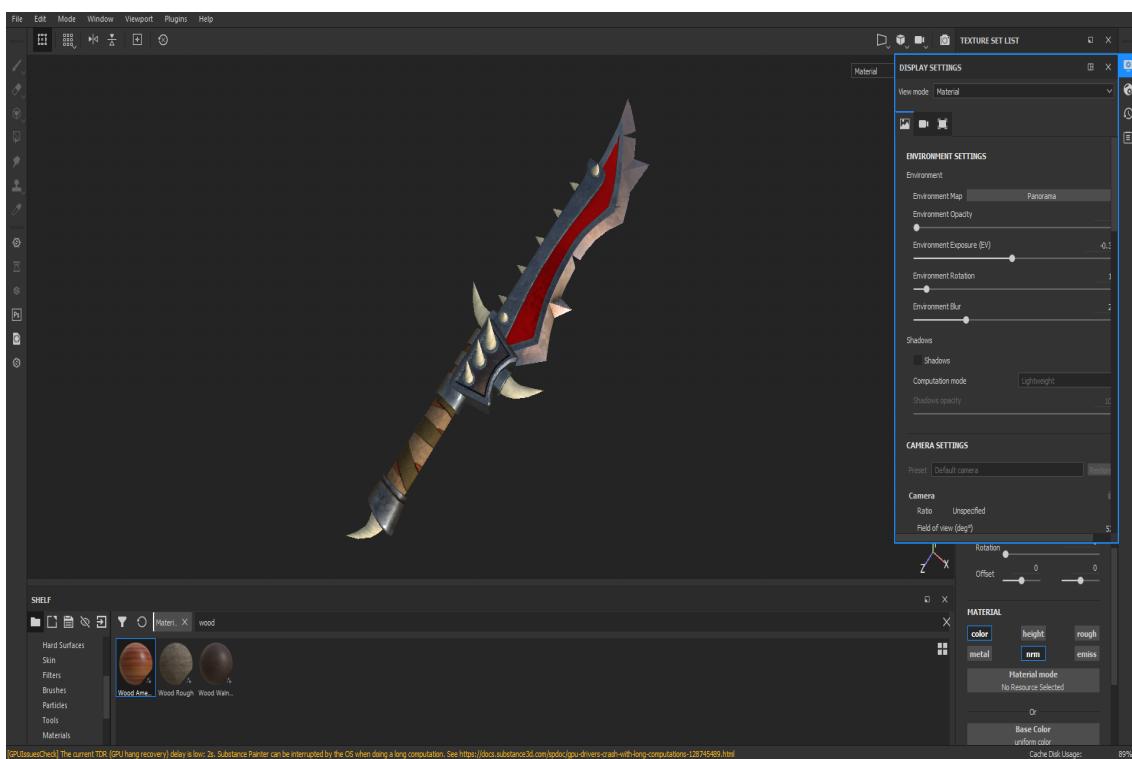


01. UV transformations

Um parâmetro que costuma se comum em todos os *smart materials* é o *UV transformation*, nele podemos alterar o tamanho dos elementos relacionados em **Scale**, normalmente a altura e largura estão vinculadas para evitar deformações, mas caso precise trabalhar cada parâmetro em separado basta desativar o botão com o cadeado presente nesta opção, também temos o **rotate** que gira os elementos em relação a UV.

EXERCÍCIO II

Faça uma texturização, utilizando um dos modelos feitos nos módulos anteriores, prepare a *mesh high poly*, use máscaras com *ID color* e *Smart materials*.



Nesse exercícios aprendemos:

- » Preparar low e high mesh para texturização.
- » Importar modelos para substance.
- » Usar máscaras.
- » Configurar smart materials.

ADD EFFECT

Apesar da biblioteca considerável de *smart materials*, ao criarmos texturas próprias precisaremos criar efeitos personalizados, como sujeira, arranhões e a simulação de determinadas superfícies, no menu de camadas existe o botão **Add Effect** onde podemos acrescentar diversos efeitos a camadas e máscaras, valendo a pena experimentar individualmente de acordo com a situação.



Add generator:

Bom para criar efeitos que afetam a superfície do objeto como desgaste ou sujeira, costuma funcionar melhor aplicado a uma máscara de uma camada de preenchimento.



Add filter:

Em sua maioria criam efeitos que determinam do que o material é feito, como tipos diferentes de metal, por exemplo. Costumam funcionar melhor aplicados a uma camada de preenchimento.



Add paint:

Adiciona uma subcamada de pintura. Quando aplicamos geradores e filtros a uma máscara esta passa a ter seu conteúdo definido pelo efeito, sendo necessário acrescentar um **paint** para fazer edições adicionais.



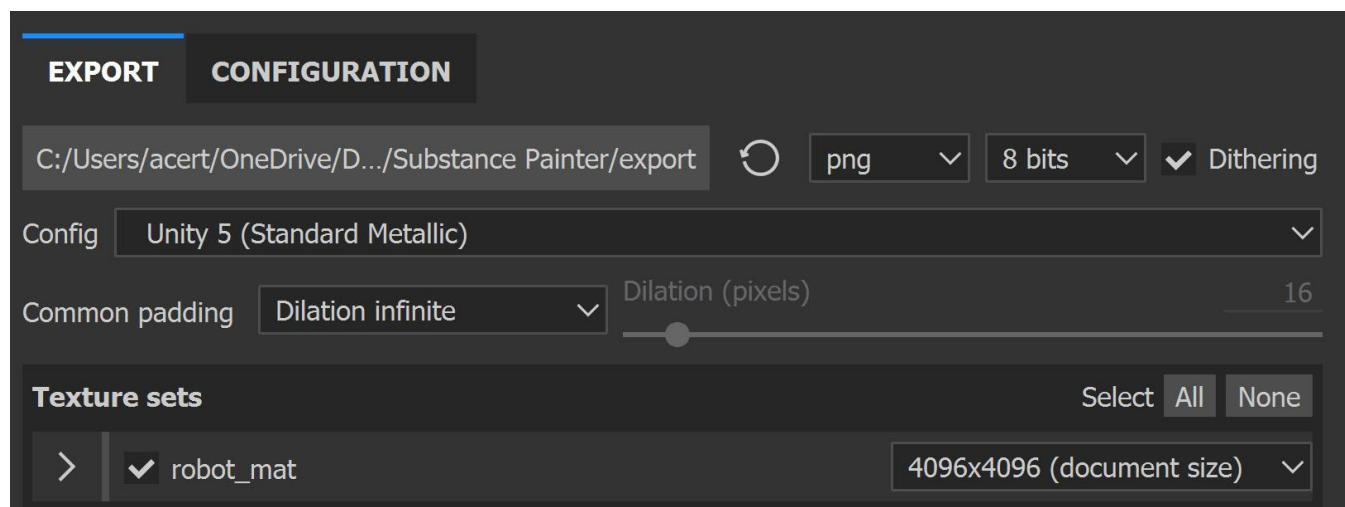
Aplicação do **+ Generator: Dirt =**

01. Criando smart materials

Depois que o efeito desejado é atingido, é possível guardar todas as configurações para aplicar em outros trabalhos criando um smart material a partir disto. É importante ter os elementos que serão agrupados dentro de uma pasta bem organizada com as layers devidamente nomeadas para facilitar a utilização futura, com tudo pronto basta clicar com o botão direito do mouse sobre a pasta e selecionar **Create Smart Material** e este novo recurso será adicionado à sua **Self**.

EXPORTAÇÃO DE TEXTURAS

Exportar o trabalho do substance para uma engine de jogos é simples, por mais que existam configurações avançadas bem específicas, no caminho **File/Export Textures** quando escolhemos no **Config** para onde vai ser feita a exportação estas opções já são configuradas automaticamente. Ainda assim há detalhes que merecem atenção como o formato de saída das imagens, sendo os mais usados **PNG** prezando pela otimização e **TARGA** pela qualidade, além do tamanho do documento, sendo que quanto maior a imagem, maior a qualidade no entanto apresenta mais custo em processamento.



Quando é feito a exportação o Substance gera imagens adequadas a **Engine** selecionada, estas imagens devem ser anexadas ao material no local de destino.



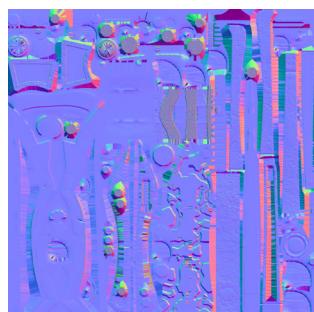
Difuse/albedo (cor)



Emissive (luz)



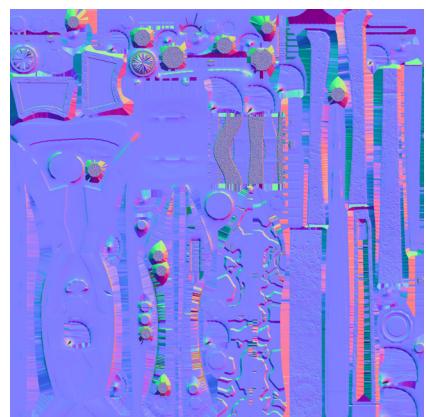
Metalic (metalico)



Normal (volume)

EXERCÍCIO III

Faça uma texturização criando seus próprios *smart materials*, faça a importação e remonte o material no unity ou em uma plataforma de portfólio de sua preferência.



Neste exercício aprendemos:

- » Organizar pastas
- » Criar smart material
- » Exportar as texturas

Este é um ótimo ponto para iniciar seu portfólio, aproveite os exercícios em que se saiu melhor, os aprimore e finalize. Desenvolver um acervo de modelos bem trabalhados é um primeiro passo para ser um profissional na área.



GAME 3D

INTRODUÇÃO

UNITY 3D

INTRODUÇÃO A PROGRAMAÇÃO

FLAPPY BIRD

MECÂNICAS DE JOGOS

TPSHOOTER

UNITY 3D - 2

INTRODUÇÃO

1. O QUE É UMA ENGINE DE JOGO, OU UM MOTOR DE JOGO?

Uma engine, ou motor, é um programa que roda jogos, isso é, renderiza frames, calcula física e roda os códigos que regram e criam as mecânicas do jogo.

- » Exemplos de engines e seus jogos:
- » Unity: Hearthstone, Rust, Escape From Tarkov, Ori and the Blind Forest.
- » Unreal: PUBG, Fortnite, ARK, Tekken 7.
- » Frostbite: Battlefield, Fifa, Need For Speed.
- » CryEngine: Crysis, Far Cry, Prey.
- » GameMaker: Undertale, Deadbolt, DeathsBolt.
- » Godot: Shipwreck.
- » RedEngine: Cyberpunk77, The Witcher.
- » RAGE: GTA, Red Dead Redemption.

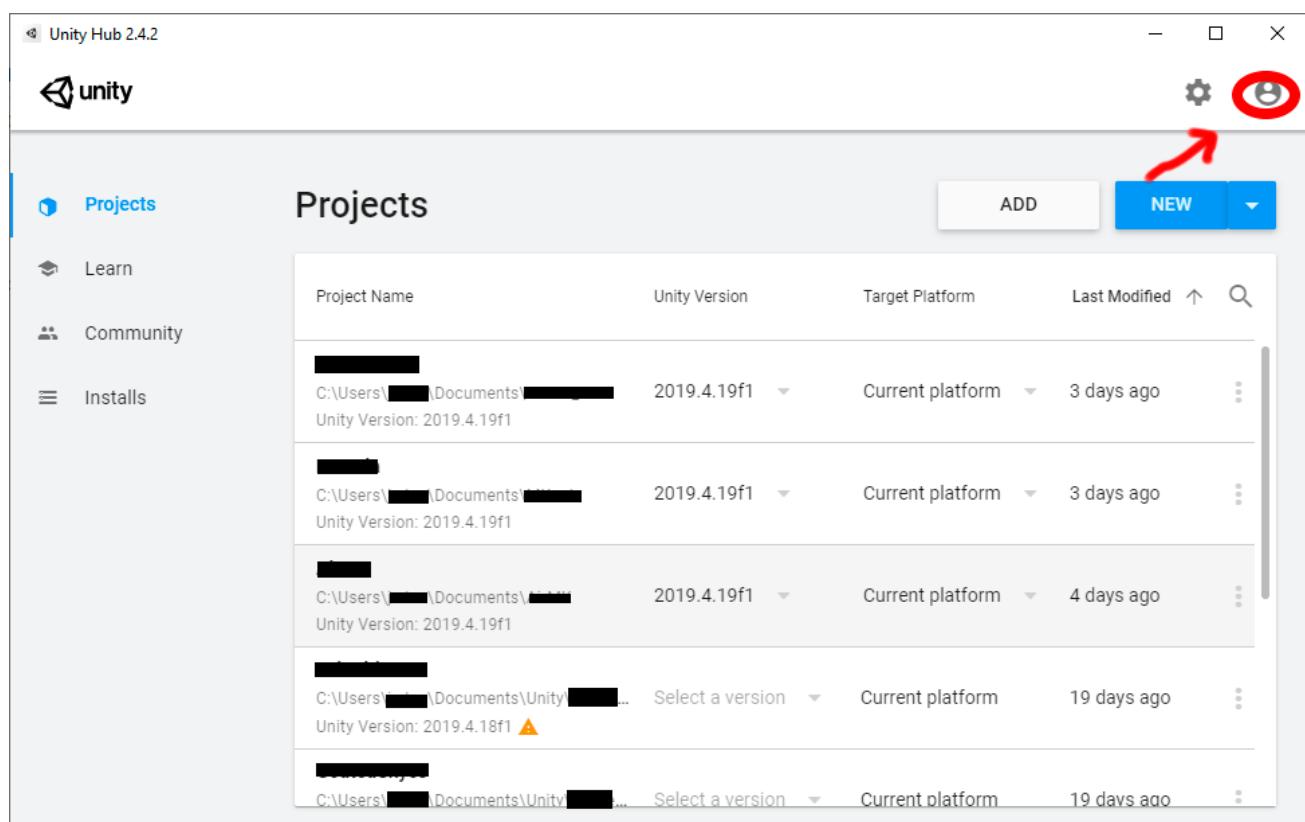
Existem outras, como Construct, Dagor, LibGDX, Cocos2D, MonoGame e muito mais! O que é necessário para produzir um jogo? Além de muito trabalho, planejamento e perseverança, são necessários alguns conhecimentos, como Game Design, Arte digital e Programação. Independente da forma que esses processos se dão, todos eles são necessários. GameDesign é formado por um conjunto de teorias que serão levemente abordadas durante a criação dos jogos deste curso. Para fins de aprendizado, iremos primeiro tentar repetir os conceitos aplicados no jogo Flappy Bird. A Arte Digital pode ser 2D ou 3D. Neste curso nosso foco será a arte 3D que aprendemos nos módulos anteriores. E a Programação será o maior foco deste módulo, tendo como meta, tornar os alunos aptos a desenvolver um jogo por completo usando a Unity Engine como ferramenta.

UNITY 3D

Nesse módulo vamos aprender sobre a Unity Engine. A unity possui duas faces, o Editor, onde o desenvolvedor trabalha e cria o jogo, e as builds, que é o motor e o jogo compilados em um aplicativo só. Em resumo, uma build é jogável e o editor é o

ambiente de desenvolvimento.
Ferramentas que vamos usar:

Ferramentas que vamos usar: O Unity Editor é um conjunto de ferramentas que pode ser usado no desenvolvimento de Jogos. Em suma, vamos aprender a usar a maioria das ferramentas naturalmente encontradas no Unity Editor, sempre tendo em mente que nem sempre é necessário usar todas essas ferramentas para fazer um jogo e que existem uma enorme quantidade de plugins que funcionam de forma modular. Usaremos também outros programas para gerar o conteúdo a ser utilizado nos jogos que desenvolvemos. Desses programas, o único que é praticamente indispensável é a ferramenta de programação Visual Studio Community, mas veremos outros como Maya, Substance e algum editor de imagem a escolha do aluno. Ao instalar a Unity, é recomendado o uso do Unity Hub para ter uma melhor experiência no gerenciamento dos seus projetos. É altamente recomendado que esteja



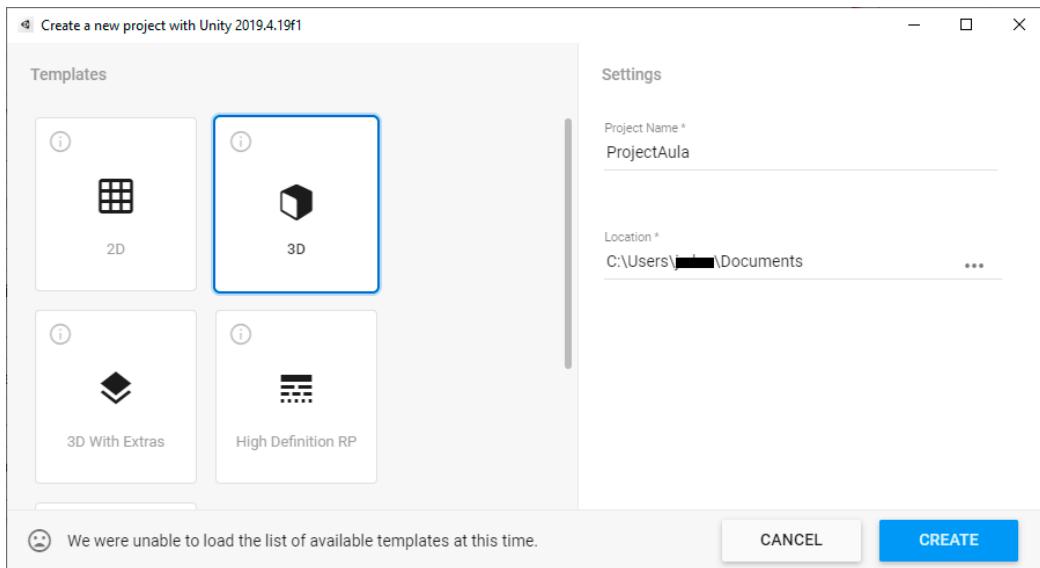
logado com sua conta Unity.

Interface do UnityHub. Área de login ressaltada. Para criar novos projetos, deve-se usar o menu a seguir, sendo que “ADD” serve para referenciar uma pasta de projeto

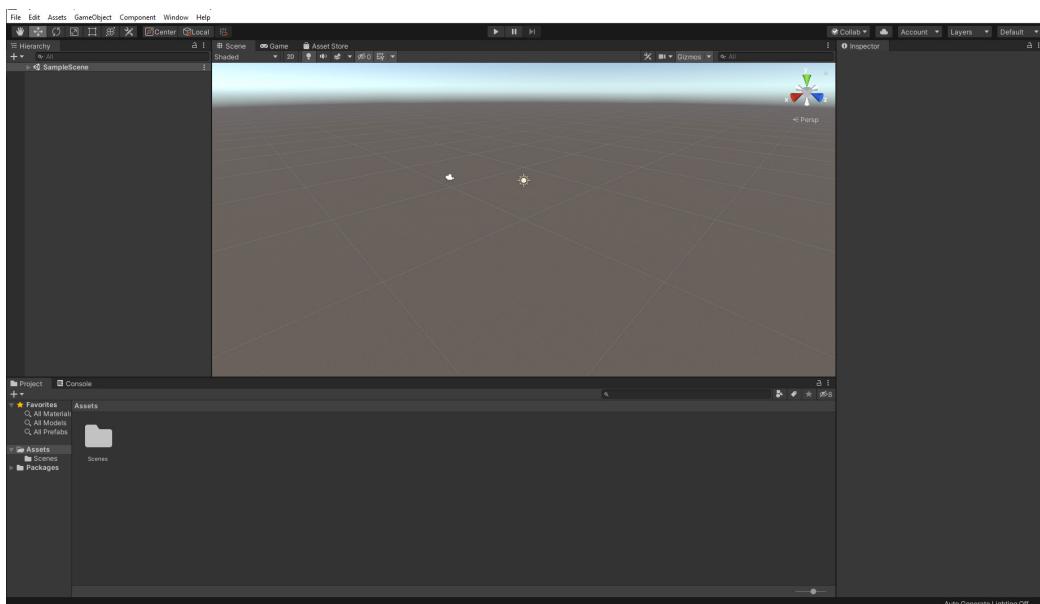


já existente e “NEW” para criar um novo projeto do zero.

Para fins de estudo, usaremos apenas o layout 3D. Preste bem atenção no local que esse projeto está sendo criado. A pasta de projeto precisa ser acessada durante o



desenvolvimento do jogo.



E após uma breve espera, o Unity Editor abrirá.
Esse é o layout padrão. É possível configurá-lo a gosto, mas o importante é conhecer



as janelas e suas ferramentas/utilidades.

Apesar de diferentes, esse segundo layout é praticamente idêntico ao primeiro, apenas separando a janela de 'game' da janela de 'scene'. Um breve resumo de cada janela e suas funções:

- » Menus: assim como na maioria dos programas, aqui ficam vários menus agrupados. Menus como salvar, abrir, preferências, janelas e muitas outras ações relacionadas ao macro contexto da unity.
- » Simulate menu: três botões. O primeiro inicia ou para a simulação do game, o segundo pausa ou despausa a simulação, e o terceiro pula um frame. Geralmente é usado com a simulação pausada. Importante saber quais mudanças feitas no projeto durante uma simulação são descartadas ao final da simulação.
- » Hierarchy: aqui ficam os 'GameObjects' organizados de forma hierárquica, isto é, objetos filhos ficam agrupados dentro dos seus pais e objetos sem parentesco ficam agrupados dentro de um objeto maior chamado de cena, um arquivo do tipo 'scene'
- » Scene: Trabalha junto com a hierarquia. Aqui é o mundo do jogo, porém possui visão totalmente desassociada da visão do jogo, que é controlada por um objeto "câmera" que está dentro da cena. Essa é uma janela de desenvolvimento, apenas, embora ela tenha uma forte associação com o jogo em si.
- » Game: é onde o jogo simulado acontece. Aqui será visto apenas o que estiver no campo de visão do objeto 'camera'. Quando a build for feita, o resultado esperado é extremamente semelhante ao resultado visto nessa janela.
- » Project: aqui ficam os 'assets' do jogo. Essa janela é como uma simulação de uma janela 'Explorer' do Windows e os arquivos vistos nela são arquivos comuns guardados na pasta 'Assets' dentro da pasta do projeto.
- » Console: aqui o código informa ao desenvolvedor o que está acontecendo. Erros, bugs, warnings e até mesmo textos descritivos são expostos, pelo código, aqui. O programador pode acessar essas funções por meio da classe 'Debug'
- » Inspector: aqui é feita a parte de controle de parâmetros do jogo e onde se controla os 'Components' de um 'GameObject' e suas variáveis. É onde a maior parte do trabalho dentro da engine deve acontecer.
- » Outros: existem várias outras janelas na unity. Naturais, plugins ou até mesmo fruto da programação desse projeto, é comum elas assumirem essa posição na engine, porém não existe nenhuma regra pra isso. Outros menus da própria, como o menu de conta da Unity, 'Collaborate' que é uma ferramenta de trabalho em equipe e de salvar na nuvem e outras janelas de configuração do ambiente de jogo são encontradas aqui inicialmente. Como foi pontuado antes, essas janelas podem mudar de posição e serem manipuladas a gosto.

Como foi pontuado antes, essas janelas podem mudar de posição e serem manipu-

ladas a gosto.

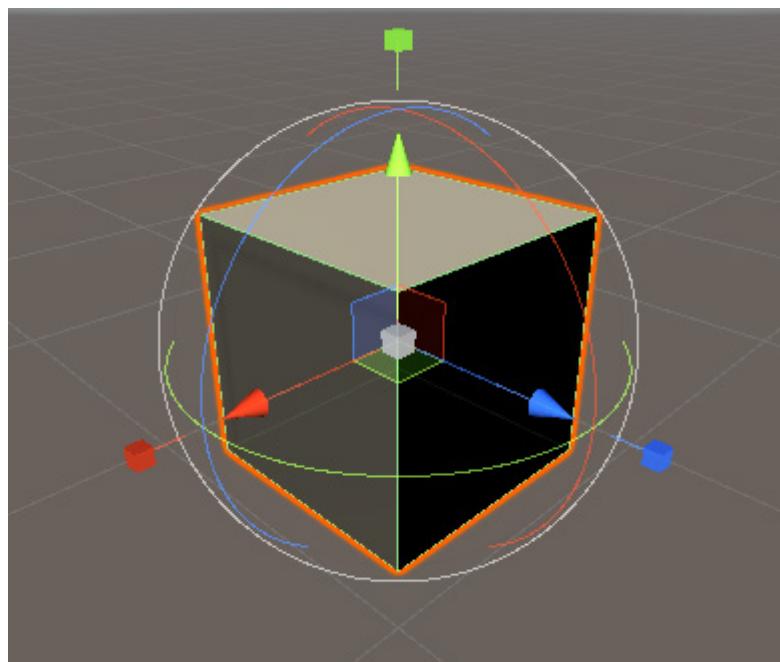
Foram citados alguns termos, que vão precisar de uma explicação:

- » Assets: São arquivos e informações usados na construção do jogo. Um modelo 3D, uma mídia de áudio, uma imagem de textura ou de sprite 2D, um arquivo de texto e até mesmo um código de programação são exemplos de assets
- » Sprite: Uma textura convertida pela unity em uma arte 2D;
- » Component: Component é um código compilado. Pode ser colocado dentro de um GameObject para que possa ser executado no jogo. Dentro da unity existem vários components por padrão que cuidam de vários comportamentos básicos, como renderizar uma sprite ou um modelo 3D, aplicar física no GameObject, uma Câmera , uma fonte de luz e muitas outras que serão apresentadas no curso. A maioria dos códigos que iremos fazer virarão components também.
- » GameObject: Um GameObject é um Objeto existente no projeto que possui, obrigatoriamente, um componente de Transform. Recebe outros components, e podem ser habilitados ou desabilitados e possuem alguns comportamentos básicos. Para fins de estudo, todo objeto de um jogo, exemplo, um cenário ou um jogador ou um emissor de partículas, é um GameObject;
- » Transform. Um component que cuida da posição, rotação e escala do seu GameObject;

Menu de Gizmos:



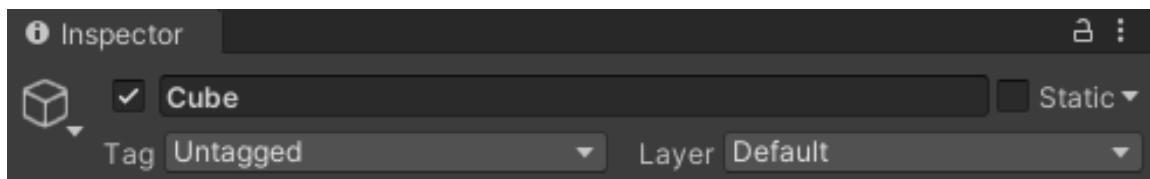
Gizmos são linhas ou curvas usadas para controlar o component Transform.



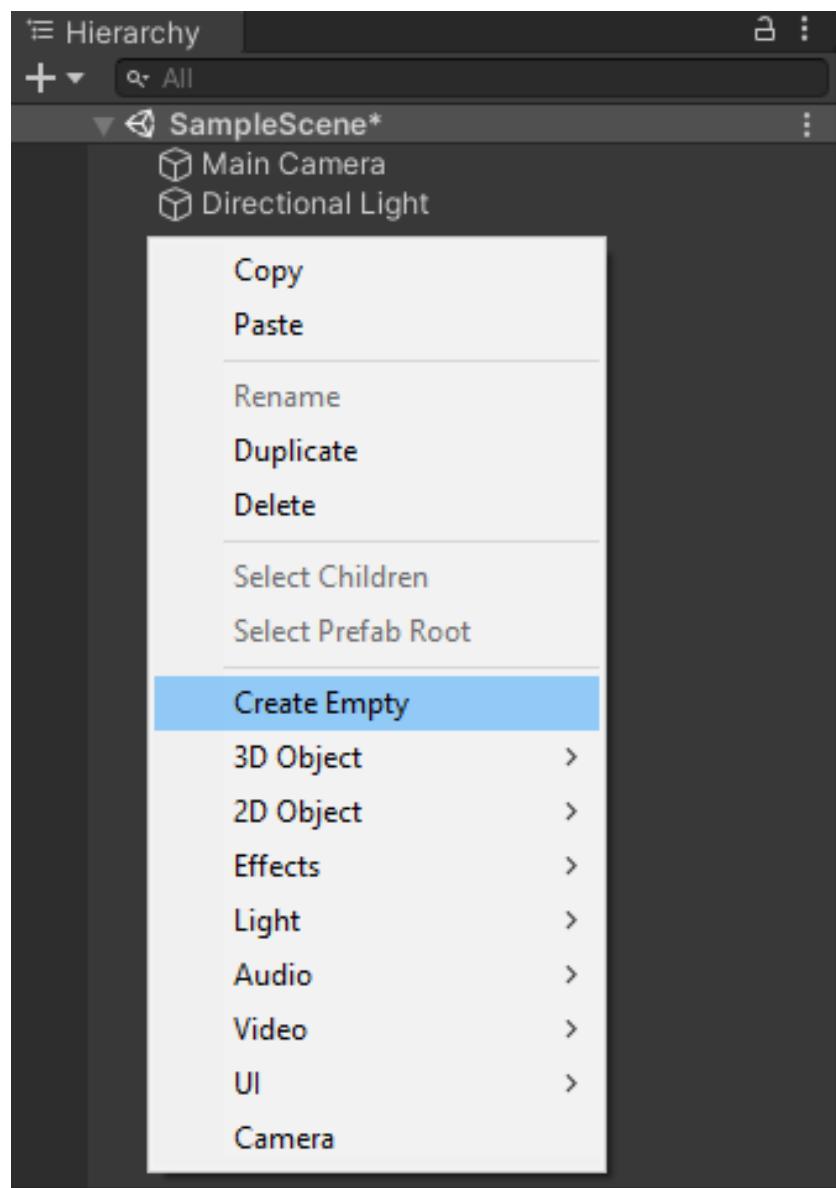
Menu auxiliar: Contém menus extras.



Menu de GameObject: contém informações comuns a todos os GameObjects, como nome do objeto, layer e tag.



Para adicionar GameObjects na cena, deve-se clicar com o botão direito na janela de hierarquia e selecionar o GameObject de escolha.

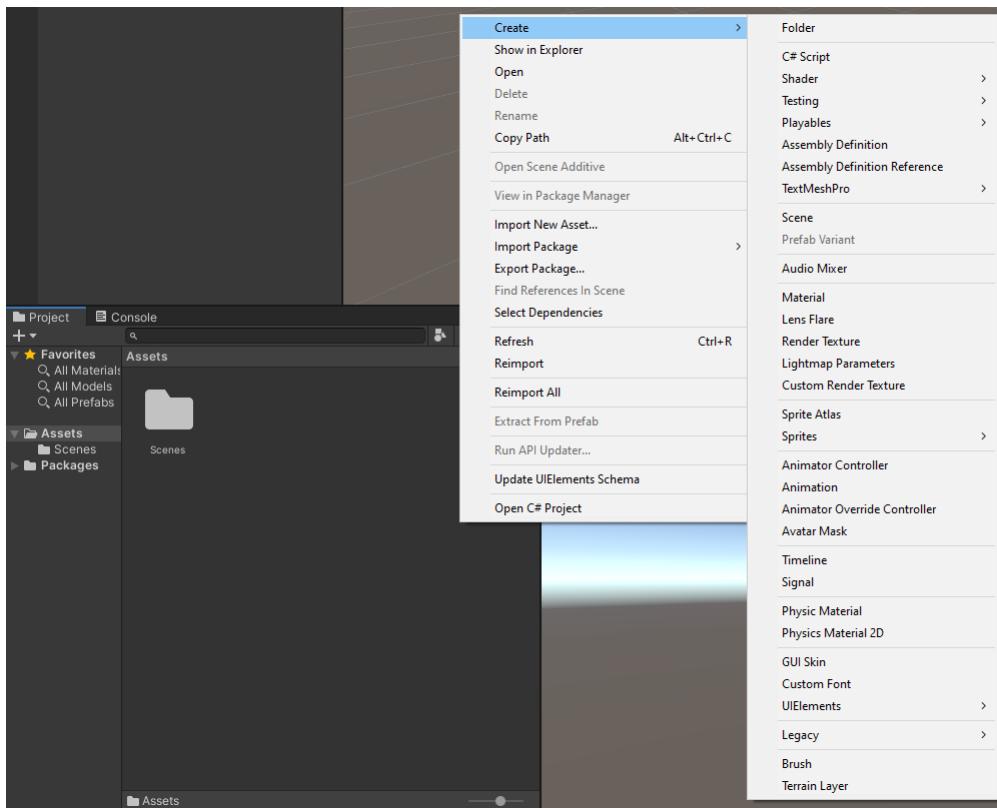


Criação de Assets:

É impossível importar como Assets, imagens, arquivos de texto ou códigos, modelos 3D, mídias de áudios e outros objetos. Esses arquivos também podem ser importados arrastando-os para o menu “Project” a partir do navegador de arquivos do computador.

Pode-se criar novos Assets a partir do menu de contexto “Create”. Códigos e cenas raramente são importados, sendo então, criados a partir do menu de contexto “Create”.

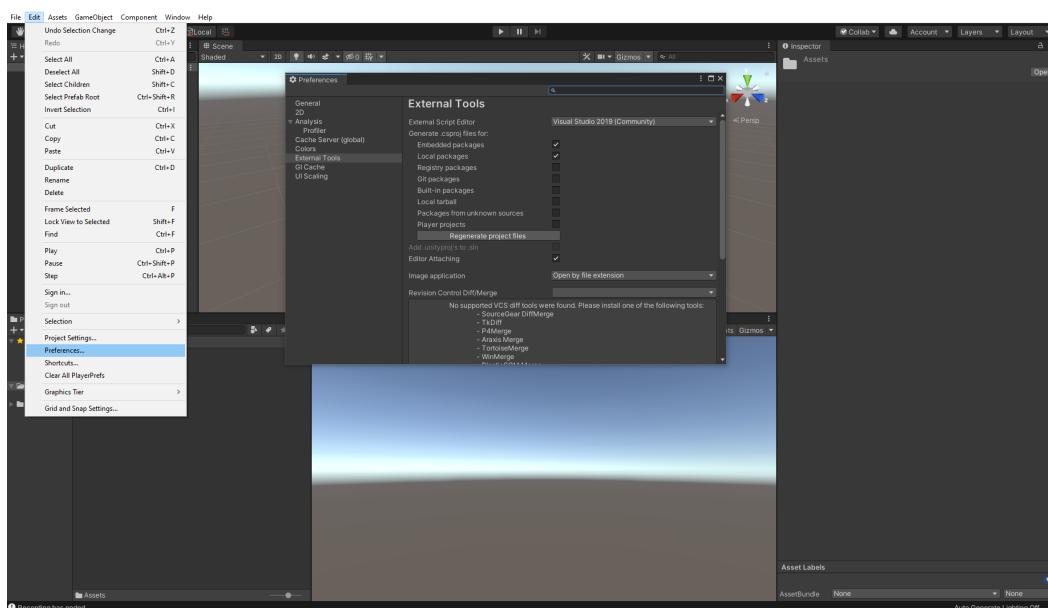
Esse menu é acessado usando o botão auxiliar do mouse dentro de uma pasta no menu “Project”



INTRODUÇÃO A PROGRAMAÇÃO

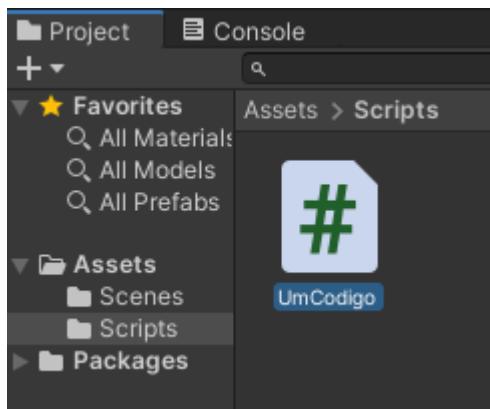
Para começar a programar na Unity, é recomendado ter acesso ao Visual Studio Community. O download do Visual Studio Community normalmente acontece junto com o download da Unity Engine por meio do Unity Hub.

Uma vez instalado a Unity Engine e o Visual Studio Community, deve-se configurar a Unity para trabalhar em conjunto com o Visual Studio. Para isso, devemos acessar o menu de contexto “Edit->Preferences”, definir o Visual Studio Community como “External Script Editor” e às vezes é necessário apertar o botão “Regenerate Project



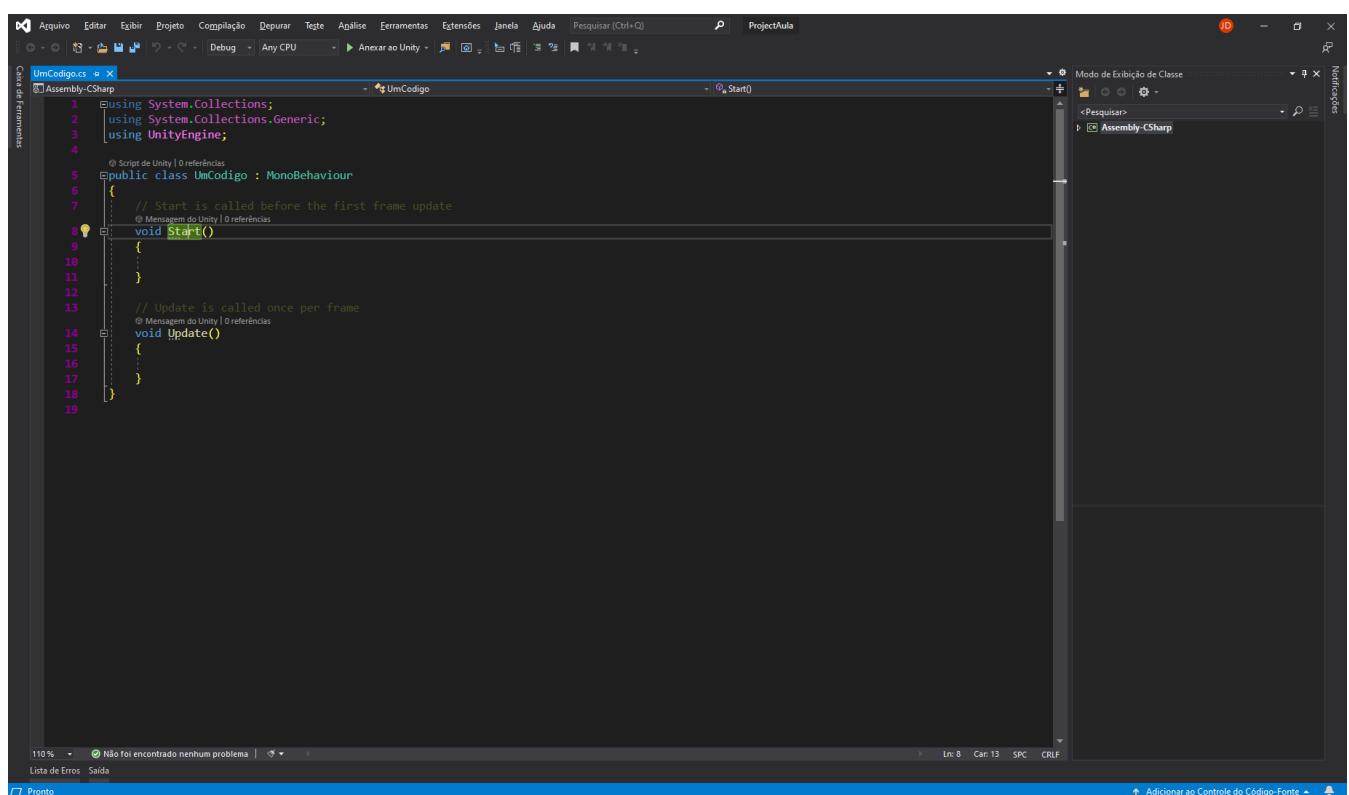
Files”.

Uma vez configurado, para acessar o Visual Studio Community a partir da Unity é



necessário clicar duas vezes em um “Script”, ou código, dentro do menu “Project”. É recomendado o uso de pastas nomeadas e bem organizadas.

Uma vez que o Visual Studios Community tenha terminado sua inicialização, o seguinte layout é esperado. É recomendado fazer login no Visual Studio Community. Vale ressaltar que “scripts” são documentos de texto simples. Portanto, o Visual Studio Community não passa de um editor de texto com inúmeras ferramentas que tornam a programação menos maçante e facilita a detecção de erros. Além de colorizar partes diferentes dos códigos, dando uma melhor percepção e visão geral do código



ao programador.

A maioria dos códigos precisam de Bibliotecas. Uma biblioteca é uma referência a um grupo de códigos existentes em algum lugar do seu computador e/ou desse projeto.

No caso do C#, declaramos as bibliotecas usando a seguinte expressão: `using NOME.DA.BIBLIOTECA`. No exemplo acima temos três bibliotecas. Duas delas fazem parte das bibliotecas System, que fazem parte do C# e uma da Unity, Essa biblioteca na Unity é a parte desse código que traz o ambiente da unity para dentro do mundo

dos códigos.

Fora isso, os outros dois outros requerimentos que a Unity pede é que o nome da classe seja o mesmo nome do código dentro do menu “Project”, dentro da Unity e que a classe “herde” de MonoBehaviour. Se algum desses requerimentos falhar, a Unity não vai conseguir compilar esse código como um “Component”, portanto esse código não poderá ser executado.

Herança é quando um código clona funções de outro código, sendo uma herança declarada nessa sintaxe: public class NOME_DA_CLASSE : CLASSE_A_SER_HERDA-DA*

Os termos “public”, “private”, “internal” e “protected” são chamados de “nível de acesso” e podem ser trocados entre si. Nesse caso não se deve colocar nada além de

```
UmCodigo.cs  ✘ public class UmCodigo : MonoBehaviour
```

public.

Também é necessário entender como um código se estrutura, ou seja, a formatação do texto. Para que o compilador, o programa que transforma texto em um programa funcional, entenda.

No caso do C# se usa {} para classes e funções, <> para tipos, () para parâmetros e grupo de retornos e [] para índices. Sempre que um desse for inicializado, ele deve terminar, com o símbolo oposto, dentro do mesmo bloco de código.

Linhas de código que não possuem fechamento automático, devem ser finalizadas com ;

Os programadores normalmente aprendem e masterizam isso metendo a mão no teclado.

Como nosso código foi criado pela Unity, pelo menu de contexto “Create”, esse código já vem preenchido. Algo assim:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class UmCodigo : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Tudo que estiver dentro do bloco de classe fará parte do nosso “Component”, que nesse caso está com o nome “UmCodigo”. Lembrem-se de criar nomes que simbolizem a função do “Component”. “UmCodigo” é um ótimo exemplo de um péssimo

nome para um “Script/Component”

Temos dois exemplos logo de cara: a função Start() e a função Update(). A função Start() acontece no primeiro frame em que esse “Component” estiver ativo. A função Update() acontece todo frame que esse “Component” estiver ativo. Se Start() e Update() acontecerem em mesmo frame, Start() acontece antes do Update().

Essas funções são herdadas de MonoBehaviour. Existem muitas outras dessas funções herdadas, além de várias variáveis/propriedades. Veja mais sobre MonoBehaviour na documentação da Unity, acessando o link: [Unity - Scripting API: MonoBehaviour](#).

A Unity é extremamente bem documentada, tanto a parte da programação, quanto a parte da manipulação do Editor. [Unity - Manual](#)

No fim das contas, programação é manipulação de informação. Os códigos manipulam essas informações e a engine lê e mostra essa informação. Basicamente, o jogo é feito de informações ordenadas de um jeito específico. Essas informações são chamadas de variáveis.

Variável é basicamente uma caixa com um nome. Essa caixa se chama “type” ou tipo. E o nome funciona como um endereço. Mantendo a analogia da caixa, o tipo seria o formado desta caixa, você não consegue, por exemplo, adicionar um carro dentro de uma caixa de celular, e seria inapropriado colocar um celular numa caixa feita para colocar um carro, embora em alguns casos seja possível fazer isso. Vale lembrar que no mundo da programação, tudo é focado em otimização, e você como programador deve focar em manter seus códigos otimizados também. Não é uma obrigação, mas é boa prática e boas práticas ajudam a criar um bom código.

Variáveis são modulares, isso é, o programador pode criar elas de acordo com a necessidade do projeto.

As variáveis básicas que são comum na maioria das linguagens de programação são:

» Inteiro: um número simples. Não possui casas decimais.

São tipos de Inteiros:

byte > vai de 0 até 255

short > vai de -32768 até 32767

int > vai de -2147483648 até 2147483647

Existem outros tipos e variantes desses tipos, mas esses são os mais comuns dos inteiros.

» Ponto Flutuante: um número com casas decimais. Seu tipo é “float”.

» Booleana: uma variável que só aceita dois valores, “true” ou “false”. Traduzindo, “verdadeiro” ou “falso”. É impressionante o potencial dessa variável no mundo da programação. Seu tipo é “bool”.

» Caractere. Uma letra, um número ou um símbolo que pode ser usado em um texto. Seu tipo é “char”

Essas variáveis são o coração dos códigos. A maioria das outras variáveis existentes usam essas variáveis de alguma forma.

Exemplos:

- » String, uma cadeira/sequência de caracteres, constitui um texto.
- » Color, quatro pontos flutuantes, um representando a cor vermelha, outro a cor verde, o terceiro representando a cor azul e o último representando a transparência. Esses valores vão de 0 a 1, sendo 0 > 0% e 1 > 100%. Os mais atentos devem ter percebido que vermelho, verde, azul => red, green, blue => RGB. que é um padrão de cores muito conhecido. Na Unity, RGBa, com 'a' de alpha ou transparência como dito antes, é o padrão de cor base.
- » Color32, a mesma coisa que o anterior, porém em vez de usar ponto flutuante, usa byte. Isto é, os valores vão de 0 a 255 sendo 0 > 0% e 255 > 100%. É mais leve e mais do que suficiente para 99% das situações. É o padrão RGB mais conhecido.
- » Os vetores, geralmente, são duplas ou trios de números. São comumente ponto flutuante ou inteiros do tipo int. Extremamente útil para se trabalhar com ambientes multidimensionais, sendo vetores duplos ideias para 2D e vetores triplos para 3D.

São tipos de Vetores:

Vector2 > composto das variáveis x e y. Ambas do tipo float;
Vector2Int > composto das variáveis x e y. Ambas do tipo int
Vector3 > composto das variáveis x, y e z. Ambas do tipo float;
Vector3Int > composto das variáveis x, y e z. Ambas do tipo int
Vector4 > composto das variáveis x, y, z e w. Ambas do tipo float;
Vector4Int > composto das variáveis x, y, z e w. Ambas do tipo int

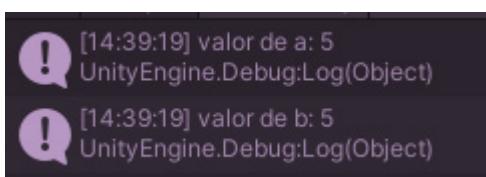
Resumindo e reforçando, variáveis são como caixas que recebem um tipo específico de informação. O programador declara seu tipo e dá um nome a ela. Esse nome é como um endereço.

Não foi bem explicitado antes, mas variáveis são usadas apenas de duas formas, ler e escrever.

Para isso a gente utiliza o operador '='. Mais pra frente veremos mais sobre operadores em geral, mas o operador '=', nomeado de "assing" ou assinalar, executa uma leitura e uma escrita ao mesmo tempo. Ele lê o valor da variável da direita e escreve o que foi lido na variável da esquerda.



```
void Start()
{
    int a = 0;
    int b = 5;
    a = b;
    Debug.Log("valor de a: " + a);
    Debug.Log("valor de b: " + b);
}
```



[14:39:19] valor de a: 5
UnityEngine.Debug:Log(Object)
[14:39:19] valor de b: 5
UnityEngine.Debug:Log(Object)

Veja que antes de assinalar b em a, foi necessário assinalar nessas variáveis algum valor inicial. Nesse caso, o 0 e o 5. Reparem também que o antigo valor de 'a' foi perdido por que valor de 'a' foi substituído por um valor igual ao de 'b'. Essa ação de ler e escrever é o coração da programação, visto que o resultado que a gente vê na tela, é basicamente uma leitura de um conjunto de variáveis.

Os operadores sempre retornam algum valor. Um retorno é uma variável descartável de leitura, que, se usada à direita de um operador '=', pode ter seu valor escrito em outra variável. O operador '=' também possui um retorno, o que torna possível essa operação:



```
void Start()
{
    int a = 4;
    int b = 5;
    int c = 0;

    c = a = b;
    Debug.Log("valor de a: " + a);
    Debug.Log("valor de b: " + b);
    Debug.Log("valor de c: " + c);
}
```

[10:11:03] valor de a: 5
UnityEngine.Debug:Log(Object)
[10:11:03] valor de b: 5
UnityEngine.Debug:Log(Object)
[10:11:03] valor de c: 5
UnityEngine.Debug:Log(Object)

Repare que, a parte de escrever do operador '=' só acontece depois da parte de leitura, sendo que a leitura só acontece quando houver apenas uma variável/retorno, isso é uma regra imutável. Nesse caso, o primeiro "assing" acontece, ele verifica que existe uma operação depois dele e passa a esperar pelo retorno dessa operação para que ele possa executar sua função de escrever, esse retorno só vai vir quando o segundo "assing" executar a leitura, nesse caso de uma variável, e a escrita, retornando o que foi escrito. Assim o primeiro "assing" pode executar sua escrita.

Repare no debug que o valor de 'b' foi passado para 'a' e para 'c', exemplificando bem o que foi explicado acima.

Abaixo fica um exemplo de como o programa entende esse comportamento. Os parênteses têm inúmeras funções na programação, mas quando se trata de ope-

```
c = (a = b);
```

radores, parênteses fazem a função de forçar a espera de um retorno específico. Isto é, dizer para o operador que antecede esse parênteses para esperar pelo valor de retorno do grupo que está entre parênteses antes de continuar sua execução.

Voltando para explicação das variáveis, seria muito trabalhoso e pouco efetivo usar variáveis que o programador colocar manualmente. É pra isso que a matemática de programação existe! E é daí que nasceram os operadores matemáticos.

Operadores matemáticos são operadores focados em aplicar cálculos matemáticos entre variáveis. Já dá para puxar um extra aqui e ressaltar que operadores em geral, trabalham direto com as variáveis. Porém a maioria dos operadores não escrevem, geralmente eles fazem a leitura do objeto a direita e da esquerda, aplicando uma conta matemática entre eles e retornando uma nova variável, de um tipo predefinido

· Operador '+' - Aplica uma soma.

Note que, como foi dito, 'a' e 'b' permanecem o mesmo valor de antes da linha de cód.

```
void Start()
{
    int a = 4;
    int b = 5;
    int c = 0;
    c = a + b;
    Debug.Log("valor de a: " + a);
    Debug.Log("valor de b: " + b);
    Debug.Log("valor de c: " + c);
}
```

! [16:47:28] valor de a: 4
UnityEngine.Debug:Log(Object)
! [16:47:28] valor de b: 5
UnityEngine.Debug:Log(Object)
! [16:47:28] valor de c: 9
UnityEngine.Debug:Log(Object)

digo que tem a operação, porém 'c', que foi criado com o valor 0 passou a ser a soma de 'a' + 'b', nesse caso, 4 + 5, logo: 9. Note também que o valor de 'c' só foi alterado por que ele está à esquerda de um operador '=' (assing). Na maioria dos casos, é necessário salvar o retorno de um operador matemático em uma variável, mas existem exceções que veremos mais pra frente.

Operador '-' - funcionamento idêntico ao '+', mas ele aplica uma subtração ao invés de uma soma.

Nesse exemplo, o valor de 'b' é maior que o de 'a', portanto resultado

```
void Start()
{
    int a = 4;
    int b = 5;
    int c = 0;
    c = a - b;
    Debug.Log("valor de a: " + a);
    Debug.Log("valor de b: " + b);
    Debug.Log("valor de c: " + c);
}
```

! [16:49:50] valor de a: 4
UnityEngine.Debug:Log(Object)
! [16:49:50] valor de b: 5
UnityEngine.Debug:Log(Object)
! [16:49:50] valor de c: -1
UnityEngine.Debug:Log(Object)

Operador '*' - aplica uma Multiplicação entre duas variáveis. A única diferença desse operador pros operadores '+' e '-' é que o operador '*' segue as regras matemáticas de ordem de execução, isso é, multiplicações e divisões ocorrem antes de soma e subtração

Operador '/' - idêntico ao operador '*', mas aplica uma divisão ao invés de uma mul-

```
void Start()
{
    int a = 4;
    int b = 5;
    int c = 0;
    c = a * b;
    Debug.Log("valor de a: " + a);
    Debug.Log("valor de b: " + b);
    Debug.Log("valor de c: " + c);
}
```

! [17:29:43] valor de a: 4
UnityEngine.Debug:Log(Object)
! [17:29:43] valor de b: 5
UnityEngine.Debug:Log(Object)
! [17:29:43] valor de c: 20
UnityEngine.Debug:Log(Object)

tiplicação.

Operador '%' — retorna o resto da divisão entre dois valores. Útil para descobrir se

```
void Start()
{
    int a = 4;
    int b = 5;
    int c ... = 0;
    c = a / b;
    Debug.Log("valor de a: " + a);
    Debug.Log("valor de b: " + b);
    Debug.Log("valor de c: " + c);
}
```

```
[17:34:14] valor de a: 4
UnityEngine.Debug:Log(Object)
[17:34:14] valor de b: 5
UnityEngine.Debug:Log(Object)
[17:34:14] valor de c: 0
UnityEngine.Debug:Log(Object)
```

um valor é par ou ímpar, para manter criar um comportamento cíclico, entre outras funções.

Reforçando: vale pontuar que todos os operadores executam no mínimo uma lei-

0%1 => 0; 1%1 => 0; 2%1 => 0; 3%1 => 0; 4%1 => 0; 5%1 => 0;	0%2 => 0; 1%2 => 1; 2%2 => 0; 3%2 => 1; 4%2 => 0; 5%2 => 1;	0%3 => 0; 1%3 => 1; 2%3 => 2; 3%3 => 0; 4%3 => 1; 5%3 => 2;	0%4 => 0; 1%4 => 1; 2%4 => 2; 3%4 => 3; 4%4 => 0; 5%4 => 1;
----------------------------------------------------------------------------	----------------------------------------------------------------------------	----------------------------------------------------------------------------	----------------------------------------------------------------------------

tura, e o operador que executa uma escrita, geralmente é o “assing”. Então, usar operadores, sem a presença do operador ‘assinalar’, não muda o valor das variáveis envolvidas. As exceções desta regra são os operadores híbridos.

Operador ‘+=’. simula o comportamento de um operador ‘=’ e um operador ‘+’.

```
int a = 1;
int b = 2;
b ... = b + a;
```

↔

```
int a = 1;
int b = 2;
b ... += a;
```

Operador ‘-=’. simula o comportamento de um operador ‘=’ e um operador ‘-’.

```
int a = 1;
int b = 2;
b ... = b - a;
```

↔

```
int a = 1;
int b = 2;
b ... -= a;
```

Operador ‘*’. simula o comportamento de um operador ‘=’ e um operador ‘*’.

```
int a = 1;
int b = 2;
b ... = b * a;
```

↔

```
int a = 1;
int b = 2;
b ... *= a;
```

Operador ‘/’. simula o comportamento de um operador ‘=’ e um operador ‘/’.

```
int a = 1;
int b = 2;
b /= a;
```

↔

```
int a = 1;
int b = 2;
b = b / a;
```

Operador '++', ou "Increment" - Soma 1 no valor base.

```
int a = 1;
a = a + 1;
a++;
```

↔

```
int a = 1;
a++;
```

↔

```
int a = 1;
a += 1;
```

Operador '--', ou "decrement" - Subtrai 1 no valor base.

```
int a = 1;
a = a - 1;
a--;
```

↔

```
int a = 1;
a--;
```

↔

```
int a = 1;
a -= 1;
```

Operador '%=' simula o comportamento de um operador '=' e um operador '%'.

```
int a = 1;
int b = 2;
b = b % a;
```

↔

```
int a = 1;
int b = 2;
b %= a;
```

O interessante é que quando não se está acostumado com programação, é comum acreditar que, ou os operadores matemáticos são tudo na programação ou que eles têm um uso muito limitado, de nicho. Tipo uma transferência em uma conta bancária ou um programa usado em caixas de mercado. A verdade é que, sempre considerando que a programação é a arte da manipulação de dados, operadores matemáticos sempre acabam tendo seu lugar de importância. Não importa se é um jogo, um caixa eletrônico ou uma inteligência artificial. É interessante entender que manipular dados quase não possuiria nenhum valor se o próprio computador/programa não pudesse comparar esses valores de alguma forma. Para isso existem os operadores booleanos, também chamados de operadores condicionais ou operadores comparativos. Primeiro vamos entender por que eles são chamados de operadores booleanos. E para isso, basta olhar pro tipo de valor que esses operadores retornam, uma variável tipo 'bool'. Como foi indicado antes, uma variável booleana é uma variável de dois estados: o estado falso e o estado positivo. A maioria das linguagens de programação possui algum tipo de variável booleana, sendo que algumas possuem tipificação numérica e outras possuem valores significativos. Isto é, umas usam 1 e 0 para representar verdadeiro ou falso, respectivamente e outras usam termos como 'true' ou 'false', que do inglês é literalmente 'verdadeiro' ou 'falso'.

No caso do C#, o tipo é 'bool' e se usa 'true' ou 'false'. O valor padrão de uma variável 'bool' é 'false'.

operador negate '!. Quando aplicado na frente de uma variável ou retorno booleana, inverte o valor do resultado. O que era 'true' passa a ser false e vice-versa; Veja como 'a' começa falso, porém o resultado no console é exibido como 'true'. Isso porque, depois da primeira chamada de console, ocorre uma negação de 'a' e um 'as-

```
void Start()
{
    bool a = false;
    Debug.Log("valor de a: " + a);
    a = !a;
    Debug.Log("valor de a: " + a);
    a = !a;
    Debug.Log("valor de a: " + a);
    a = !a;
    Debug.Log("valor de a: " + a);
}
```

!	[12:04:17] valor de a: False UnityEngine.Debug:Log(Object)
!	[12:04:17] valor de a: True UnityEngine.Debug:Log(Object)
!	[12:04:17] valor de a: False UnityEngine.Debug:Log(Object)
!	[12:04:17] valor de a: True UnityEngine.Debug:Log(Object)

sinalar' desse retorno no próprio 'a' antes da segunda chamada do console'. Esse processo se repete algumas vezes para melhor exemplificar o uso do operador.

Operador "equals" '=='. Uma das maiores confusões na cabeça de programadores de primeira viagem. O "equals", ou igual, é o operador de comparação direta. Retorna 'true' se, e apenas se, as duas variáveis possuem o mesmo valor.

Operador "not equals" '!='. Funciona exatamente como o operador "equals", mas o resultado vem invertido. Pela tradução, estaríamos verificado se x não é igual a y. Algo

```
void Start()
{
    int a = 4;
    int b = 5;
    int c = 5;
    bool aEb = (a == b);
    bool aEc = (a == c);
    bool bEc = (b == c);
    Debug.Log("valor de a: " + a);
    Debug.Log("valor de b: " + b);
    Debug.Log("valor de c: " + c);
    Debug.Log("a == b: " + aEb);
    Debug.Log("a == c: " + aEc);
    Debug.Log("b == c: " + bEc);
}
```

!	[11:08:34] valor de a: 4 UnityEngine.Debug:Log(Object)
!	[11:08:34] valor de b: 5 UnityEngine.Debug:Log(Object)
!	[11:08:34] valor de c: 5 UnityEngine.Debug:Log(Object)
!	[11:08:34] a == b: False UnityEngine.Debug:Log(Object)
!	[11:08:34] a == c: False UnityEngine.Debug:Log(Object)
!	[11:08:34] b == c: True UnityEngine.Debug:Log(Object)

interessante desse operador é a mistura na sintaxe dele, é um misto de operador “negate” com operador “equals”.

Além desses, existem os operadores booleanos comparativos. São usados entre dois números como uma pergunta. São eles:

- » “Maior que” ou “Greater than” ou “>”. O valor da esquerda é maior que o da direita?
- » “Maior ou igual a” ou “Greater or Equals” ou “>=”. O valor da esquerda é maior ou igual ao valor da direita?
- » “Menor que” ou “Lesser than” ou “<”. O valor da esquerda é menor que o da direita?
- » “Menor ou igual a” ou “Lesser or Equals” ou “<=”. O valor da esquerda é menor ou igual ao valor da direita?

Condicionais. Como foi falado antes, o código deve ser capaz de mudar seu comportamento de acordo com algumas condições do projeto. Existe uma solução para isso,

```
void Start()
{
    int a = 1;
    int b = 2;
    int c = 3;

    //declarado sem valor inicial
    bool result;

    result = a > b;
    Debug.Log("1 > 2: " + result);
    result = a >= b;
    Debug.Log("1 >= 2: " + result);

    result = (a + b) > c;
    Debug.Log("(1 + 2) > 3: " + result);
    result = (a + b) >= c;
    Debug.Log("(1 + 2) >= 3: " + result);

    result = a < b;
    Debug.Log("1 < 2: " + result);
    result = a <= b;
    Debug.Log("1 <= 2: " + result);

    result = (a + b) < c;
    Debug.Log("(1 + 2) < 3: " + result);
    result = (a + b) <= c;
    Debug.Log("(1 + 2) <= 3: " + result);
}
```

!	[23:27:36] 1 > 2: False	UnityEngine.Debug:Log(Object)
!	[23:27:36] 1 >= 2: False	UnityEngine.Debug:Log(Object)
!	[23:27:36] (1 + 2) > 3: False	UnityEngine.Debug:Log(Object)
!	[23:27:36] (1 + 2) >= 3: True	UnityEngine.Debug:Log(Object)
!	[23:27:36] 1 < 2: True	UnityEngine.Debug:Log(Object)
!	[23:27:36] 1 <= 2: True	UnityEngine.Debug:Log(Object)
!	[23:27:36] (1 + 2) < 3: False	UnityEngine.Debug:Log(Object)
!	[23:27:36] (1 + 2) <= 3: True	UnityEngine.Debug:Log(Object)

a condicional “if” e seus cooperadores “else” e “else if” Traduzindo, “if” significa “se” e “else” significa “senão”.

Esses termos podem autorizar ou negar a execução de trechos de código atrelados a eles.

A sintaxe é a seguinte:

```
if (booleana)
{
    //Código A
}
else
{
    //Código B
}
```

O código A só irá acontecer se booleana tiver o valor “true” e o código B só irá acontecer se booleana tiver o valor “false”. Lembre-se sempre que a booleana precisa ter seu valor lido, isso é, pode ser uma variável ou um retorno, desde que o tipo da variável seja “bool”.

O termo “else if” é uma mistura dos dois termos anteriores, agrupando o funcionamento de ambos.

```
if (booleana)
{
    //Código A
}
else if(booleana2)
{
    //Código B
}
else
{
    //Código C
}
```

Se a primeira das condições não for satisfeita, então a segunda condição será conferida e se satisfeita, então código B será executado. Caso contrário, então o código C será executado.

Com essa ferramenta, podemos controlar inúmeras situações. Considere que quando o jogador aperta a tecla W no teclado, uma booleana chamada “moverParaFrente” tem seu valor mudado para “true” e muda para “false” assim que o jogador solta ela. Agora imagine que exista um “if” conferindo o valor dessa variável e no caso dela ser “true”, o avatar do jogador deve andar para frente. Faça isso para cada uma das teclas de movimento de jogo e a movimentação direcional do jogador está pronta! Iremos ver essa e outras utilidades para as condicionais durante esse módulo.

Outra ferramenta básica da programação são os loops.

Existem vários tipos de loops, os básicos do C# são:

- » While
- » Do While
- » For
- » Foreach

O “While”, cuja tradução é ‘Enquanto’ e o “Do While”, cuja tradução é ‘Faça Enquanto’ funcionam basicamente da mesma forma. A diferença está na escrita. Considerando que `a++` é a mesma coisa que `a = a + 1`. Esse é o operador “increment”. É possível prever que esse código vai repetir 100 vezes e o valor de `a` vai ser 100;

```
int a = 0;
bool booleana = true;
while (booleana)
{
    a++;
    booleana = a <= 100;
}
```

Repare que ‘booleana’ começa com o valor inicial “true”. Se ela não tivesse esse valor inicial, esse loop não iria nem começar, portanto a será 0.

Para evitar essa necessidade de um valor inicial, foi criado o “Do While”. Independente do valor inicial de booleana, esse loop será executado.

Muito cuidado com “While”, pois esses loops são a maior causa de loop infinito que

```
int a = 0;
bool booleana;
do
{
    a++;
    booleana = a <= 100;
}
while (booleana);
```

existe. Um loop infinito pode causar inúmeros tipos de problema. Se um loop infinito for executado na simulação da Unity, sua unity irá parar de responder e será necessário usar o gerenciador de tarefas para finalizar os processos da Unity. Muitas vezes isso é um bug inofensivo, mas é bom evitar.

O loop “for” é um loop com um contador natural.

É comumente usado para percorrer arrays, listas, strings e até imagens ou sistemas de grid em um jogo.

Nesse exemplo, o código dentro do bloco também executará 100 vezes e a variável a terminará no valor 100.

```
int a = 0;
for (int i = 0; i <= 100; i++)
{
    a = i;
}
```

Apesar do resultado semelhante, “While” e “Do While” abrangem muitas outras situações. Enquanto o “For” tem essa forma de percorrer uma sequência de valores ordenada por uma expressão matemática. Nesse caso um simples operador “increment” da variável “i”.

Esse tipo de comportamento é muito conveniente na hora de fazer buscas ou afetar um grupo de variáveis específicas.

O “Foreach” tem um uso mais específico e é melhor explicá-lo quando a necessidade/opportunidade aparecer.

FLAPPY BIRD

- » Iniciar um projeto 3D simples.
- » Configurar Visual Studio Community.
- » Configurar Collab.
- » Escolher um modelo 3D para ser nosso pássaro. Pode ser uma nave no espaço, um pássaro/jato no céu ou um peixe no oceano. O que importa é que deve ser algo que faça sentido.
- » Colocar um Objeto vazio na cena para ser nosso Player, e resetar seu Transform. Esse objeto deve ter como filho o objeto 3D escolhido acima. Deve ter todos os materiais necessários devidamente configurados.
- » A malhar dentro do Player deve ter sua posição (0,0,0) e deve apontar (direção simbólica do objeto) para a direita.
- » A câmera do jogo deve permanecer em perspectiva e a luz direcional não pode criar sombras.
- » Resetar o Transform da câmera e colocar ela na posição (0,0,-10)
- » Escolher uma imagem de fundo com as características “tileable” e “seamless” na horizontal
- » Adicionar um “Quad” para ser usado como backGround, ele deve ficar na posição e (0,0,5).
- » Ajustar a escala do backGround até que ele ocupe toda a câmera
- » Tornar o backGround filho direto da câmera.
- » Adicionar um rigidbody no Player. Testar o funcionamento do rigidbody.
- » Tornar esse rigidbody kinematic.
- » Criar um script para controlar o Player.

01. Criar variável float “force”.
 02. Criar variável Rigidbody “rigidbody”
 03. Criar variável bool “hasStarted”
 04. Fazer o rigidbody se tornar dinâmico somente se o jogador já tiver apertado a tecla “space”. Use a variável “hasStarted”
 05. Criar um if() dentro do Update() que quando apertado a tecla “space” aplique uma força no rigidbody. Essa força deve ter a direção “Vector3.Up”, seu valor deve ser “force” ajustado pelo “deltaTime” e o tipo da força deve ser “impulse”.
 06. Se necessário, criar um booster para aumentar essa força quando a velocidade do rigidbody no eixo vertical for menor que 0;
- » Na Unity, acessar o menu de contexto “Edit > Project Settings > Physics” e alterar a gravidade para a (0,-80,0) ou algo próximo.
- » No Player, altere a “Force” para 200 ou algo próximo.
- » Adicionar um SphereCollider no Player.
- » Adicionar 4 BoxColliders no BackGround, criando paredes invisíveis ao redor do jogador, na fronteira da área de visão da Câmera. Um Collider para cada fronteira no eixo (X,Y). Esses colliders devem ter a mesma posição em Z que o player.
- » Criar a tag “Obstaculo”.
- » Adicionar a tag “Obstaculo” nos Colliders criados como paredes.
- » No Script do Player
01. Criar uma variável global public static bool “isDead” e dar o valor inicial de “false”.
 02. Adicionar função herdada “OnCollisionEnter”
 03. Criar uma condição que se o Player colidir com um objeto cuja tag seja “Obstaculo”, deve-se alterar o valor da variável “isDead” para “true”.
 04. Adicionar na primeira linha do update uma condição que se a variável “isDead” for “true”, deve-se mudar rigidbody.kinematic para false e retornar vazio.
- » Criar um GameObject, esse será o nosso Obstáculo.
- » Adicionar um Cylinder como filho do Obstáculo.
- » Adicionar a tag “Obstaculo” no Obstáculo.
- » Criar um script de movimentação do Obstáculo.
01. Esse script terá sua velocidade comandada por uma variável do Game Manager
- » Transformar esse Obstáculo em um Prefab e deletar o original da cena.
- » Criar um GameObject vazio e ele será nosso Game Manager.
- » Criar um Script para o Game Manager.

01. Criar uma variável GameObject chamada de obstáculo.
02. Criar uma variável public static float gameSpeed e dar a ela um valor de 3f ou algo semelhante.
03. Deve-se programar um comportamento de criação dos obstáculos a cada x segundos sendo x uma variável.
04. Deve ter um obstáculo acima e um abaixo, deve ter um espaçamento mínimo entre os dois obstáculos.
05. Sempre que o jogador passar de um grupo de obstáculos, deve-se contar um ponto pro jogador.

- » Na Unity, Criar um Canvas com a pontuação do jogador.
- » Fazer essa pontuação atualizando a cada vez que esse valor for alterado.
- » Fazer um polimento geral, e enriquecer o jogo em geral.

MECÂNICAS DE JOGOS

Muito foi falado aqui sobre Funções. Funções ou métodos são grupos de código. Esses grupos de códigos trabalham em cima de parâmetros, que são nada mais do que variáveis de uso interno de uma função. Interessante lembrar que usualmente um parâmetro tem o comportamento oposto ao de um retorno, sendo então, uma variável descartável, prioritariamente usada para leitura, mas seu foco está em passar uma informação a fim de gerar um retorno.

Existem meios de passar um parâmetro com uma chamada de saída. Isto é, antes desse código retornar algum valor, ele deve antes escrever algo neste parâmetro e esse valor persiste.

Existe também a passagem de parâmetro como referência. Isto é, o valor desse parâmetro é persistente, sendo ele escrito ou não.

Toda função ou método tem um retorno, embora na maioria das vezes esse retorno seja do tipo void. Isto é, retorno sem nenhuma informação.

Muitas coisas podem virar um método: conferir se o jogador atingiu algum inimigo com sua espada, o gatilho de uma arma, o pulo do jogador, limpar ou adicionar um item em um inventário... Basicamente qualquer código que se repita muitas vezes, em situações diferentes, deve estar dentro de um método.

Métodos que dentro de si possuem uma chamada a si mesmo são chamados de métodos recursivos. Esses métodos podem causar loop infinito se não houver uma condição acessível que possua uma chamada de retorno.

Propriedades são a mistura de variáveis com funções. A parte “get” de uma propriedade é a parte que será lida. Possui obrigatoriamente um retorno. A parte “set” é a parte da escrita. O valor a ser escrito é chamado de “value”.

A parte “set” da propriedade pode ter um nível de acesso diferente do nível de acesso da propriedade em si. Veremos exemplos mais à frente.

```

int vidaAtual = 10;
0 referências
int VidaAtual
{
    get
    {
        return vidaAtual;
    }
    set
    {
        if (value < 0)
        {
            vidaAtual = 0;
        }
        else if (value > vidaMaxima)
        {
            vidaAtual = vidaMaxima;
        }
        else
        {
            vidaAtual = value;
        }
    }
}

int vidaMaxima = 21;
0 referências
float porcentagemDeVida
{
    get
    {
        float aux = vidaAtual / (float)vidaMaxima;
        return aux;
    }
}

```

Outra coisa legal desse exemplo dentro da propriedade “VidaAtual”, na parte “set”, é que existe uma cadeia de condicionais feita para definir o valor mínimo e o valor máximo de uma variável. A biblioteca de matemática da Unity, a “Mathf” que vem junto com a biblioteca “UnityEngine” possui um método para isso, o Método “Clamp”.

Classes são corpos maiores que variáveis ou propriedades. Geralmente uma classe é um conjunto de variáveis, propriedades e funções. Elas exercem vários tipos de comportamentos. Na Unity, funcionam como Components, como um grupo de variáveis

```

int VidaAtual
{
    get
    {
        return vidaAtual;
    }
    set
    {
        vidaAtual = Mathf.Clamp(value, 0, vidaMaxima);
    }
}

```

e ou como uma central de rápido acesso a alguns métodos e variáveis. O mais importante sobre classes para um programador iniciante, é entender a existência delas e como conversar com elas, visto que a própria engine já oferece um conjunto gigantesco de Classes e Métodos que satisfazem boa parte da necessidade de um jogo. Time, Debug, GameObject, Physics, Transform, Rigidbody e UmCodigo são exemplos de classes. Repare que todos possuem a mesma cor, embora Transform e UmCodigo estejam em caráter de redundância.

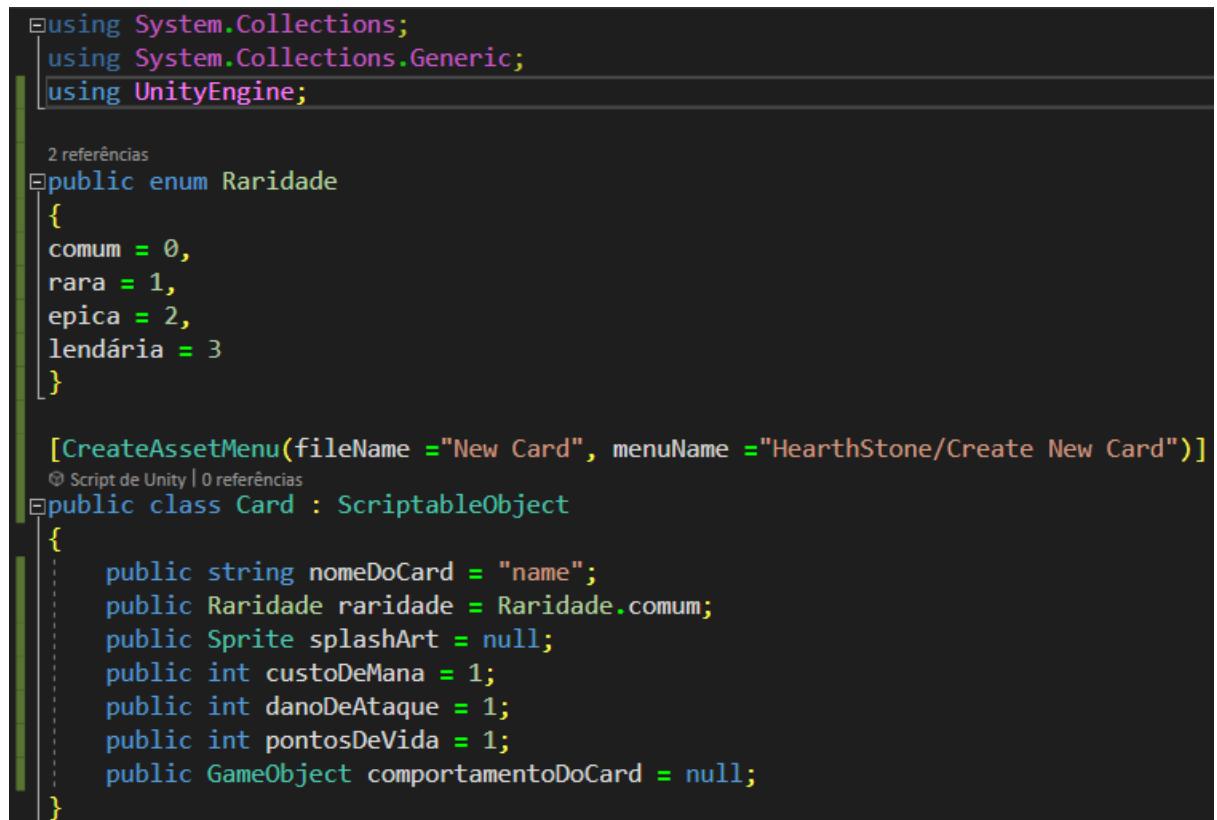
```
float a = Time.deltaTime;
Debug.Log("Hello World");
var go = GameObject.Find("Main Camera");
Physics.Raycast(transform.position, transform.forward, 10f);
Transform.FindObjectOfType<Rigidbody>();
UmCodigo.Destroy(this);
```

Um dos tipos de classes da Unity é o ScriptableObject. Tem a mesma sintaxe e exigências de um “Component”, porém a herança é da classe “ScriptableObject”. Um ScriptableObject é usado como container de informações. Um dos melhores exemplos disso são as cartas do hearthstone, onde custo de mana, dano base, vida base, splash art e raridade são informações guardadas em um ScriptableObject, que depois são lidas e instanciadas no jogo como uma carta.

Segue um exemplo de como isso poderia ser feito:

Repare na linha a seguir:

```
“[CreateAssetMenu(fileName =”New Card”, menuName =”HearthStone/Create New Card”)]”
```

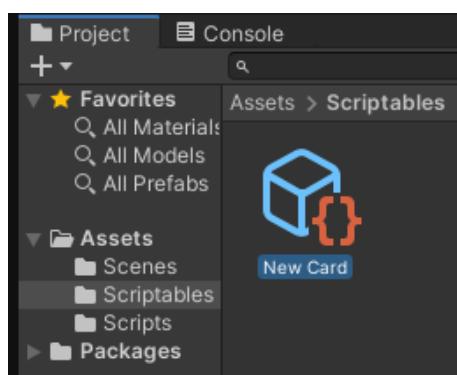
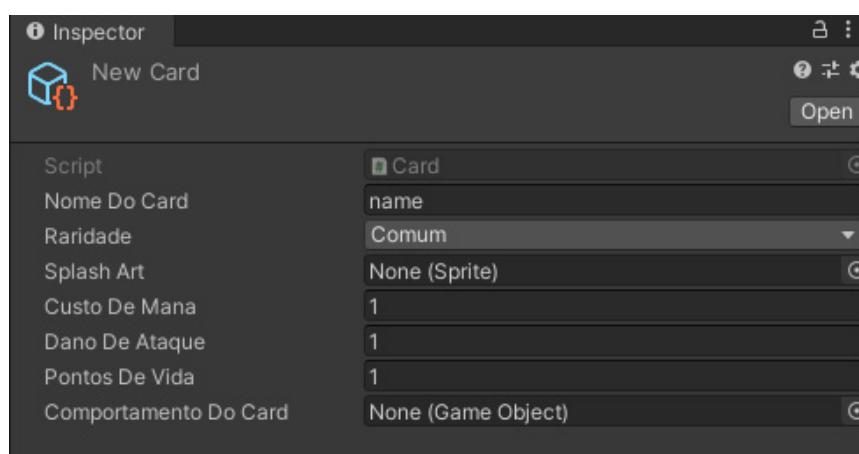
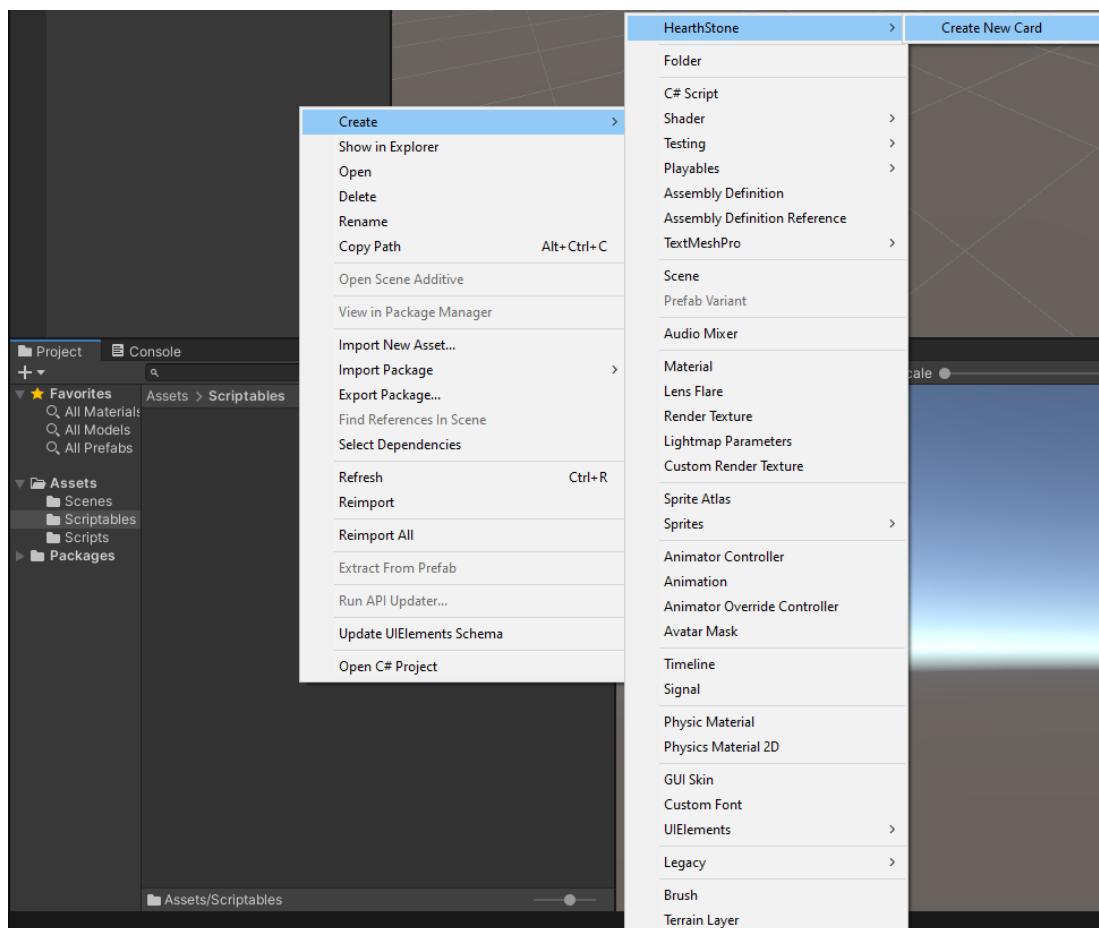


```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

2 referências
public enum Raridade
{
    comum = 0,
    rara = 1,
    epica = 2,
    lendária = 3
}

[CreateAssetMenu(fileName =”New Card”, menuName =”HearthStone/Create New Card”)]
public class Card : ScriptableObject
{
    public string nomeDoCard = “name”;
    public Raridade raridade = Raridade.comum;
    public Sprite splashArt = null;
    public int custoDeMana = 1;
    public int danoDeAtaque = 1;
    public int pontosDeVida = 1;
    public GameObject comportamentoDoCard = null;
}
```

Elas permitem a criação através do menu de contexto “Create” dentro do menu “Project”. Podemos criar vários desse, cada um com informações relacionadas a uma carta específica. E nesse caso, será terceirizado para um GameObject, e seus Components, o comportamento da carta.



Uma outra coisa que esse exemplo nos mostra é o enumerador. Em resumo, um enumerador é uma forma de associar números a nomes e vice-versa.

Vamos usar esse exemplo do hearthstone e das raridades das cartas. Cartas comuns não possuem a joia no centro, cartas raras possuem a joia azul, cartas épicas a joia roxa e cartas lendárias, a joia laranja.

```
public enum Raridade
{
    comum = 0,
    rara = 1,
    epica = 2,
    lendária = 3
}
```

Para esse exercício vamos ignorar a arte da carta, e iremos trabalhar apenas com as joias

Sabendo que a joia é um SpriteRenderer, podemos fazer o seguinte:

Isso é uma cadeia simples de condicional. Não existe nenhum tipo de complexidade nessa lógica porque a checagem só pode ter um resultado e ela é totalmente linear. A partir do momento que uma dessas condições for satisfeita, o restante das condi-



ções são ignoradas.

```
[SerializeField] SpriteRenderer joia = null;

0 referências
void DefinirJoia(Raridade _raridade)
{
    if (_raridade == Raridade.comum)
    {
        joia.gameObject.SetActive(false);
    }
    else if (_raridade == Raridade.rara)
    {
        joia.gameObject.SetActive(true);
        joia.color = Color.blue;
    }
    else if (_raridade == Raridade.epica)
    {
        joia.gameObject.SetActive(true);
        //Acha a cor Roxa, sabendo que ele se encontra entre o azul e o vermelho, nesse caso a 33% do caminho.
        joia.color = Color.Lerp(Color.blue, Color.red, 0.33f);
    }
    else if (_raridade == Raridade.lendária)
    {
        joia.gameObject.SetActive(true);
        //cria o laranja a partir de uma mistura de 100% de vermelho e 40% de verde
        joia.color = new Color(1f, 0.4f, 0f, 1f);
    }
}
```

Esse tipo de cadeia de condicional é muito comum, porém nesse exemplo ele é considerado inapropriado. Confira o exemplo abaixo:

O termo “switch” representa uma cadeia de ifs, sendo que o termo “case” define a informação a ser comparada com o parâmetro passado dentro do termo “switch”. O motivo de se usar switch ao invés de uma cadeia de condicionais é a facilidade que ele oferece no caso da necessidade de uma manutenção no código. Sem contar que

```
[SerializeField] SpriteRenderer joia = null;

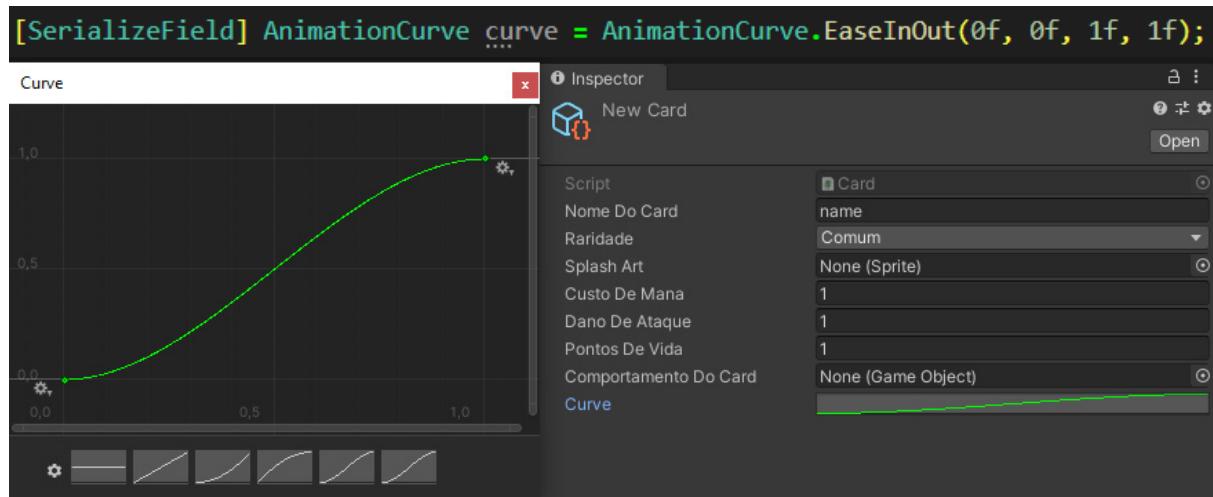
0 referências
void DefinirJoia(Raridade _raridade)
{
    switch (raridade)
    {
        case Raridade.comum:
            joia.gameObject.SetActive(false);
            break;
        case Raridade.rara:
            joia.gameObject.SetActive(true);
            joia.color = Color.blue;
            break;
        case Raridade.epica:
            joia.gameObject.SetActive(true);
            //Acha a cor Roxa, sabendo que ele se encontra entre o azul e o vermelho, nesse caso a 33% do caminho.
            joia.color = Color.Lerp(Color.blue, Color.red, 0.33f);
            break;
        case Raridade.lendária:
            joia.gameObject.SetActive(true);
            //cria o laranja a partir de uma mistura de 100% de vermelho e 40% de verde
            joia.color = new Color(1f, 0.4f, 0f, 1f);
            break;
        default:
            break;
    }
}
```

ele é mais especializado e menos convoluto.

Usar uma cadeia de condicionais para fazer esse tipo de checagem é como usar um while para percorrer uma array. Funciona normalmente, mas existe uma outra ferramenta especializada nisso e que possui alguns extras que podem e vão ser úteis.

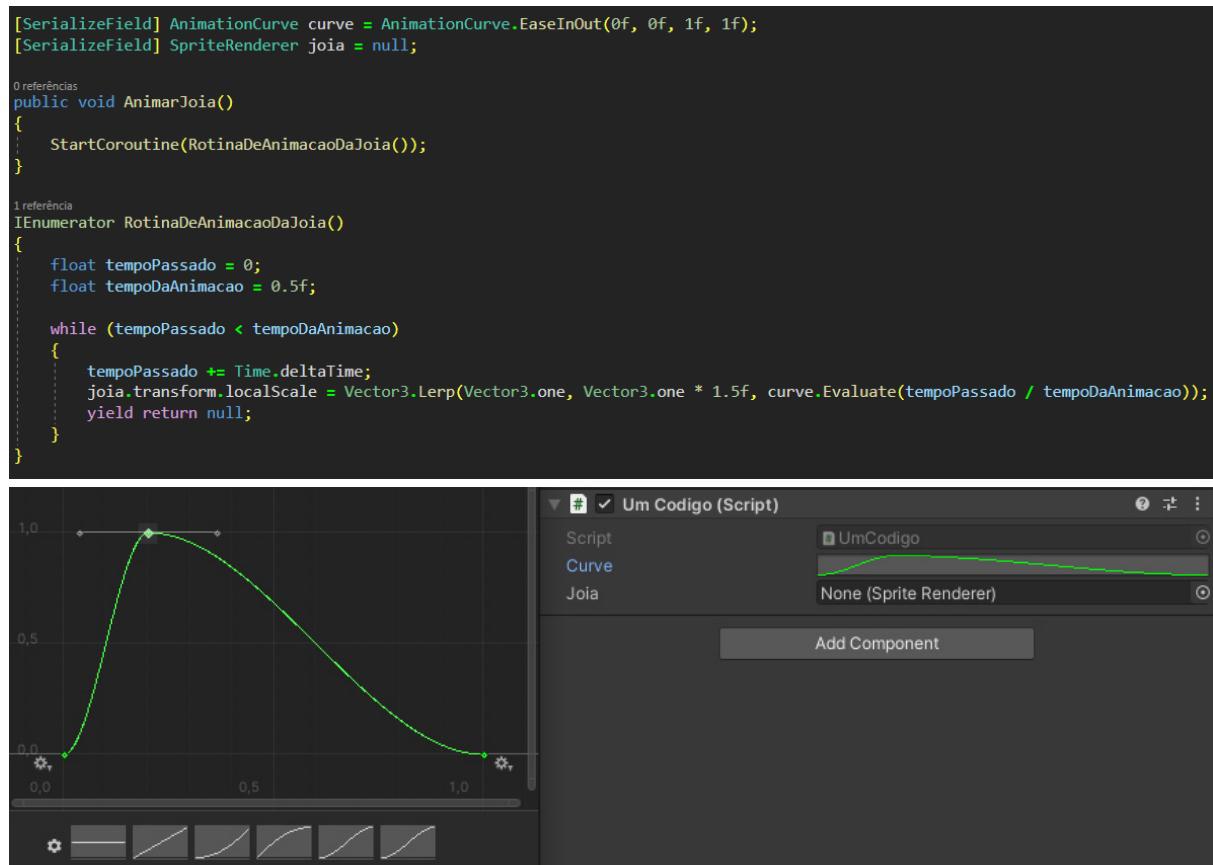
Outra ferramenta de extrema utilidade são as variáveis de curva. A mais usada na Unity é a “AnimationCurve” que embora tenha esse nome, seu valor na programação de mecânicas e feedbacks é imensurável.

É difícil explicar sua função sem um exemplo. Mas curvas são muito utilizadas em todas as áreas da computação. É uma forma de trabalhar com informações num plano cartesiano sem a necessidade de funções matemáticas. Para jogos, sua função pode ser descrita como a capacidade de transformar uma progressão linear em uma progressão manualmente configurável.



Vamos dar um exemplo usando a joia da carta do hearthstone. Imagine que se o jogador clicar na jóia, ela deverá fazer uma animação simples.

Iremos colocar um Component Button na jóia e quando o jogador ativar esse Button. A partir do clique, esse Button vai chamar essa função AnimarJoia(). Essa função começa uma Coroutine, nesse caso, RotinaDeAnimacaoDaJoia(). Uma Coroutine é um trecho de código que não está mais diretamente atrelado a essa sequência de código.



Todo IEnumerator, o método que será afetado pela Coroutine, requer a presença de no mínimo um termo “yield return”. Esse termo chama uma pausa na sequência do código dentro do IEnumerator. No exemplo acima foi mostrado um “yield return null”. Esse termo causa a pausa deste código até o final desse frame. No próximo frame, a Unity encontrará um bom momento para recomeçar esse trecho de código, e continuará do exato ponto em que foi pausado, mantendo inclusive o valor das variáveis internas (variáveis externas/globais podem ser alteradas e podem afetar o funcionamento desse trecho de código)

Existem outros tipos de “yield return”, sendo os 3 mais usados além do null, o WaitForSeconds, o WaitForSecondsRealtime e o StartCoroutine().

- » WaitForSeconds: pausa o trecho de código por uma quantidade x de segundos, sendo que esse tempo considera o relógio da simulação. Isto é, o relógio do computador multiplicado pelo Time.timeScale.
- » WaitForSecondsRealtime: mesma coisa do anterior, porém a contagem de segundos considera o relógio do computador, e não da simulação.
- » StartCoroutine: você pode começar uma Coroutine dentro de outra Coroutine. Usar isso em conjunto com um “yield return” fará o trecho de código ser pausado enquanto a outra Coroutine não terminar.

O exemplo acima cria um ciclo que durará “tempoDaAnimacao” segundos, mudando o valor da localScale da joia. Essa alteração não será linear por causa da curva.

Vamos fazer uma simulação:

Repare bem que o tempo é linear, porém o resultado da curva é variado e não possui linearidade alguma nesse caso. Sabendo que esse valor resultante será o valor usado para definir a localScale da joia, sendo que quando for 0, a local scale será Vector3.one e quando for 1 será (Vector3.one * 1.5f). Dá para interpretar essa simulação. No período de 0 segundo a 0.1 segundo, esse a localScale vai crescer rapidamente até alcançar o valor máximo desse exemplo. A partir daí irá começar a decrescer lentamente até retornar ao valor inicial, aos 0.5 segundo.

tempoPassado	tempo De Animacao	Curva em (*100)%	Resultado da Curva
0.0	0.5	0.0/0.5 = 0	0f
0.05	0.5	0.05 / 0.5 = 0.1	~0.5f
0.1	0.5	0.1/0.5 = 0.2	1f
0.2	0.5	0.2/0.5 = 0.4	~0.85f
0.25	0.5	0.25/0.5 = 0.5	~0.7f
0.3	0.5	0.3/0.5 = 0.6	~0.5f
0.4	0.5	0.4/0.5 = 0.8	~0.15f
0.5	0.5	0.5/0.5 = 1	0f

Como tudo nesse módulo, é mostrado um único exemplo de uso. No caso, de AnimationCurve e Coroutine. A criatividade e a necessidade são as maiores ferramentas de um programador. Achar novas maneiras de usar essas ferramentas não é divertido, como é uma habilidade que pode te colocar na frente dos seus colegas programadores.

Agora que essas ferramentas foram entregues é preciso pô-las em prática para que elas possam se sedimentar como conhecimento.

Mas antes disso é necessário reforçar o conhecimento sobre Herança de código.

Imaginem, por exemplo, quais são os controles básicos de uma arma de fogo que um jogador pode ter em um jogo de FPS tipo CS:GO ou Valorant.

- » Disparo
- » Disparo secundário
- » Recarregar

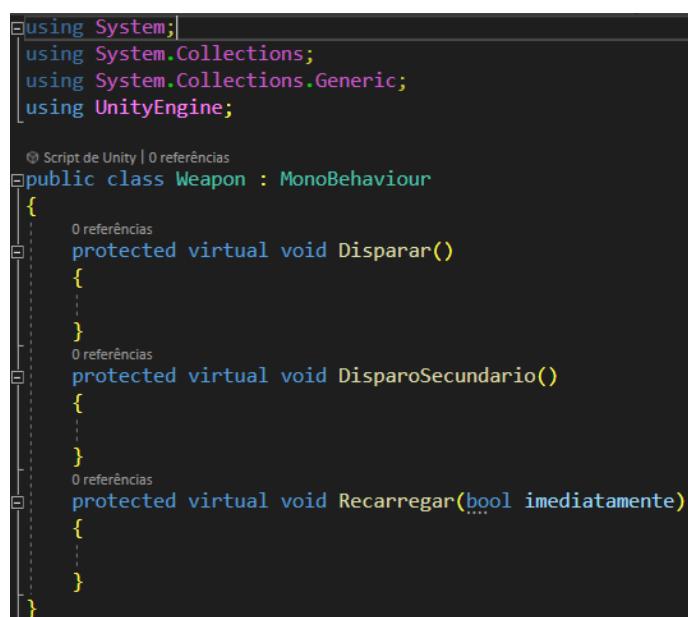
Com isso é possível fazer um jogo de tiro! Mas esse jogo ficaria bem repetitivo se todas as armas tivessem somente esses comportamentos. Esses são comportamentos básicos, é esperado que toda arma tenha algo do gênero.

É possível criar um Component que faz o controle do jogador. Isto é, quando o jogador aperta o botão do mouse, o component chama a função Disparo(). Quando o jogador aperta o botão auxiliar, o Component chama a função DisparoSecundario(). Quando o jogador aperta a tecla 'R' o Component chama a função Recarregar(). Agora é possível que uma arma se diferencie da outra, por exemplo. Uma possui uma mira, portanto o DisparoSecundario() deve abrir/fechar a mira. A outra pode trocar o comportamento do Disparo() toda vez que o DisparoSecundario() for chamado. Uma terceira arma pode fazer um disparo múltiplo e a outra disparar um lança granadas no seu DisparoSecundario().

Todas elas têm o mesmo controle. O que muda é o que está dentro do código na função DisparoSecundario(). Podemos então fazer um Component para cada arma, sendo que esses Components herdarão do Component Weapon. Veja que a herança do MonoBehaviour não foi perdida, estamos herdando de uma classe que herda de MonoBehaviour.

Repare que o nível de acesso, no caso o protected, é o mesmo na classe pai e na classe herdeira. O termo “virtual” contido na classe Weapon é o que permite o termo “override” sobrescrever o corpo da função.

Esse foi um exemplo bem superficial. Heranças são usadas das mais diversas formas e no nosso projeto TPS Shooter teremos vários exemplos.



```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Weapon : MonoBehaviour
{
    protected virtual void Disparar()
    {
    }

    protected virtual void DisparoSecundario()
    {
    }

    protected virtual void Recarregar(bool imediatamente)
    {
    }
}
```

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

Script de Unity | 0 referências
public class ScopedWeapon : Weapon
{
    protected override void DisparoSecundario()
    {
        AbrirOuFecharMira();
    }

    private void AbrirOuFecharMira()
    {
        // Fazer Abrir ou fechar a mira
    }
}

```

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

Script de Unity | 0 referências
public class WeaponWithGrenadeLauncher : Weapon
{
    protected override void DisparoSecundario()
    {
        TentarAtirarGranada();
    }

    private void TentarAtirarGranada()
    {
        // Tentar atirar granada se ainda tiver munição
    }
}

```

Falando em Shooters, é legal falar também de reaproveitamento de objetos no jogo. Conhecida com Pooling, a técnica de reaproveitamento de objetos vem da necessidade de se otimizar um jogo ao máximo. Pooling é um termo que traduzido de forma leviana seria “piscinando”. Leva o pensamento de que se você tira um objeto, usa ele e depois que foi usado, você joga ele numa piscina cheia de outros objetos do mesmo tipo e quando você precisar de um desses, você pega qualquer um deles

e bota em uso. É meio confuso, porém é uma técnica muito simples, que diminui o trabalho de um dos maiores vilões da otimização de jogos, o Garbage Collector. Simplificando, ao invés de você criar um novo objeto toda vez que você precisar de um, você pega um que já foi usado antes. Porém agora está desocupado, lhe dá o status de ocupado e o usa. E em vez de destruir um objeto no final do seu uso, apenas o esconde e o torna desocupado. Um exemplo disso seria projéteis em um shooter. Dificilmente num shooter terão mais de 20-30 projetos ativos num mesmo período de tempo. Embora não seja difícil que algumas centenas de disparos ocorram em poucos minutos. Em vez de criar centenas de projéteis, cria-se 30, por exemplo, e usa uma lógica de pooling em cima desses 30.

A melhor forma de fazer isso é usando listas.

Os termos “Count”, “Contains”, “Remove”, “RemoveAt” e “Add” são métodos da classe List.<>.

» Count (Contagem): o tanto de objetos dentro dessa lista.



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ProjectilePooling : MonoBehaviour
{
    [SerializeField] GameObject prefabDoProjetil = null;
    List<GameObject> emUso = new List<GameObject>();
    List<GameObject> parado = new List<GameObject>();

    public GameObject PegarProjetilParado()
    {
        if (parado.Count > 0)
        {
            GameObject projetilEscolhido = parado[0];
            parado.RemoveAt(0);
            emUso.Add(projetilEscolhido);
            return projetilEscolhido;
        }
        else
        {
            GameObject novoProjetil = Instantiate(prefabDoProjetil, transform);
            emUso.Add(novoProjetil);
            return novoProjetil;
        }
    }

    public void RetornarProjetil(GameObject projetil)
    {
        if (emUso.Contains(projetil))
        {
            emUso.Remove(projetil);
            parado.Add(projetil);
        }
    }
}
```

» Contains (Contém): retorna uma booleana se o valor passado estiver dentro da lista.

» Remove (Remover): remove da lista o objeto idêntico ao valor passado.

»

» RemoveAt (Remover Ali): remove da lista o valor que estiver na posição passada.

» Add. (Adicionar): colocar esse valor dentro da lista, na última posição.

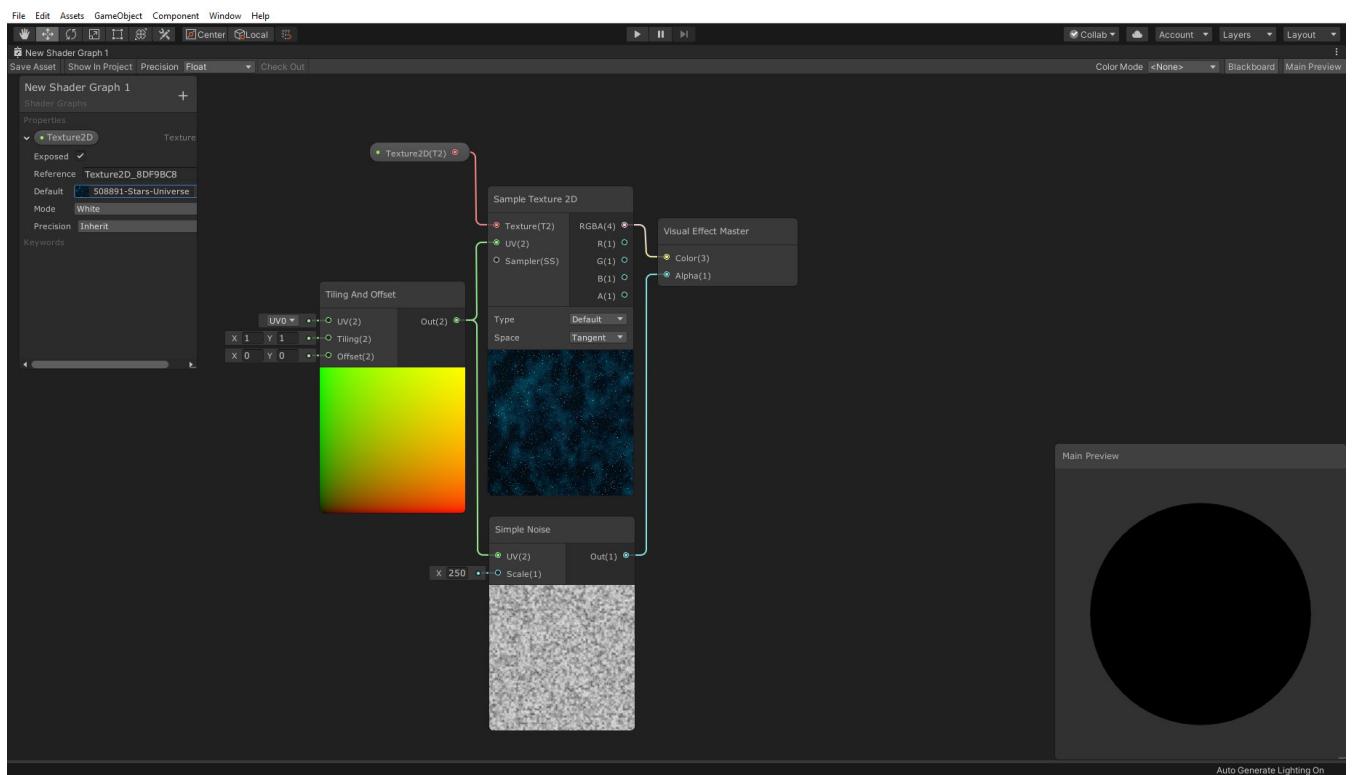
Repare que nesse caso, se não houver objetos desocupados, esse código está preparado para criar novas cópias de “prefabDoProjétil”. Sempre que esse projétil terminar sua rotina, ele, ou algum gerenciador de projéteis deve retornar esse projétil usando o método RetornarProjétil(), passando o projétil como parâmetro.

TPSHOOTER

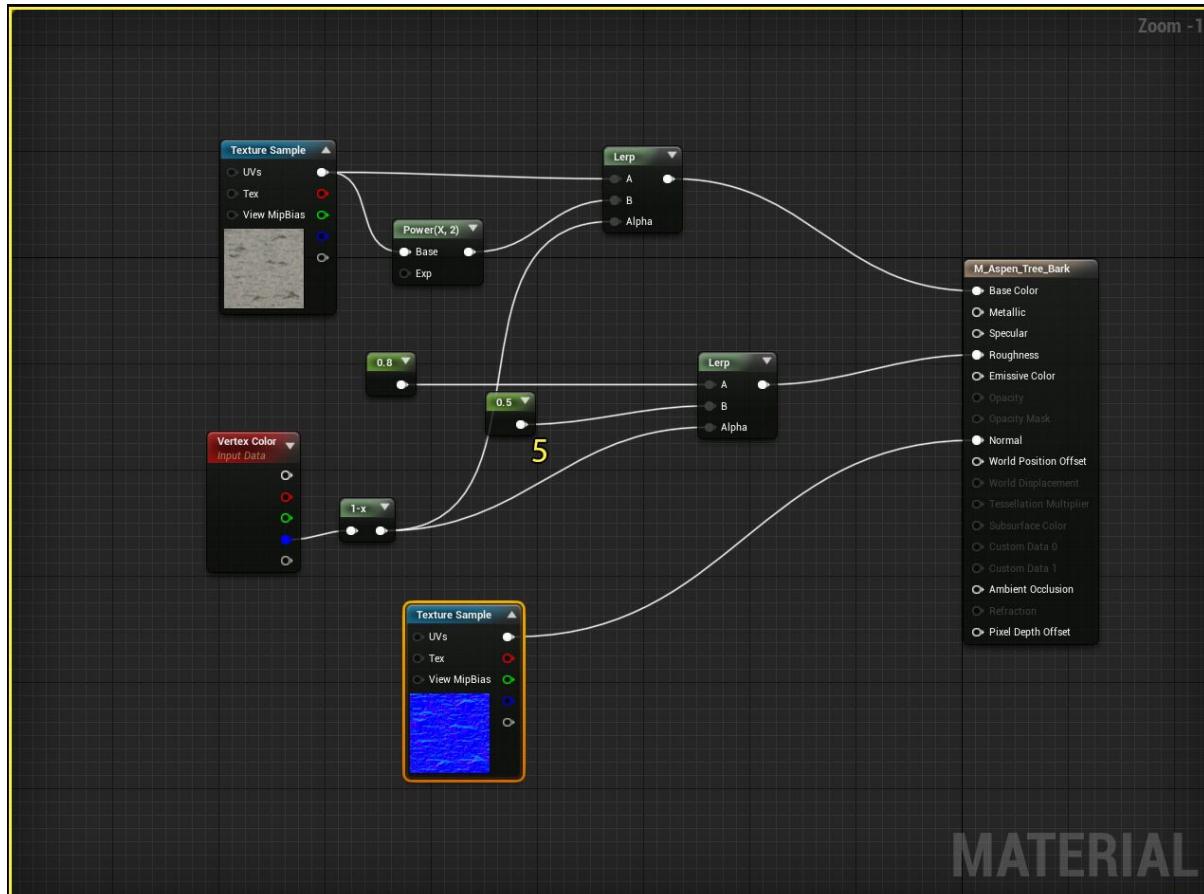
- » Criar um cenário usando formas simples. Onde o jogador e os inimigos forem se movimentar deve ser plano e possuir várias obstruções, desde que não bloqueie a passagem por completo.
- » Personagem deve ser um humanóide e possuir um rig e animações de TPS.
- » Criar um animator para o Player e criar todas as transições de movimento.
- » Programar a movimentação do personagem.
- » Programar uma arma simples para o jogador.
- » Programar um Spawner de inimigos que ativa com a proximidade.
- » Criar inimigos, com animator.
- » Programar um comportamento básico para inimigos.
- » Fazer projéteis causar dano nos inimigos.
- » Programar derrota de inimigos.
- » Criar várias armas.
- » Programar um sistema de armas usando herança e scriptables.
- » Criar power ups que trocam a arma do jogador e dão buffs, como velocidade de movimento, cura e dano extra
- » Expandir o mapa, aumentando a quantidade de inimigos e até programando inimigos mais difíceis.
- » Criar UI, menu e polir a gameplay.

UNITY 2

A Unity é muito versátil. Não somente ela é cheia de ferramentas embutidas, ela te dá tudo que você precisa para fazer novas ferramentas dentro dela. Para fazer isso, é necessário usar a biblioteca “UnityEditor” e usar Heranças específicas. No caso, é possível criar “Inspectors” personalizados para “Components” e “ScriptablesObjects” ou criar Janelas especializadas para fazer alguma tarefa específica. Esse tema é gigantesco e é uma área auxiliar no desenvolvimento de jogos. Porém, é extremamente importante saber que essas ferramentas existem. Confira o link:[Unity - Scripting API](#): e navegue pela área de “UnityEditor”. Essas ferramentas podem se tornar plugins. Um plugin é um conjunto de ferramentas extras, feitas por terceiros e este, incrementa a experiência de um desenvolvedor. A Unity oferece a AssetStore como um meio de oferecer conteúdo extra, inclusive plugins. Muitas das ferramentas que hoje são naturais da Unity, foram plugins que por serem tão poderosos, acabaram sendo absorvidos pela engine. Alguns exemplos: TextMeshPro, ProBuilder, PostProcessing e ShaderGraph. Você encontra essas ferramentas na janela de “Package Manager”. ShaderGraph é uma ferramenta de edição de shaders. Shaders são as funções matemáticas que cuidam da renderização de objetos no jogo. Essa matemática pode ser manipulada para criar vários efeitos. Até poucos anos atrás, era preciso saber uma programação super complexa e confusa. Hoje essa programação ainda é usada, mas em vez de programar manualmente, é possível usar uma programação visual super intuitiva para criar esses efeitos.



É uma ferramenta super poderosa e vai fazer seu jogo ficar muito mais bonito! Existem outras ferramentas de programação visual de shaders, a mais famosa na unity é o ShaderForge. Outro exemplo é a UnrealEngine, que trabalha muito bem esse tipo de ferramenta.



O interessante desse assunto, é saber que o programador pode criar sistemas para facilitar e acelerar o desenvolvimento do jogo. Além disso, é possível usar plugins de terceiros que vão economizar horas ou semanas de trabalho, além de possibilitar mecânicas ou efeitos visuais que em outras situações, seriam praticamente impossíveis dado a experiência dos desenvolvedores e o tempo do projeto. Estudar, pesquisar e estar aberto a conhecer mais sobre o ambiente de trabalho na engine é uma das habilidades mais recompensadoras na área de desenvolvimento de jogos.



ACADEMY
DIGITAL ENTERTAINMENT