07

Configurando o projeto e movimentando o inimigo

Porque usar um game engine?

A maior dificuldade enfrentada quando queremos começar a desenvolver jogos é justamente descobrir por onde começar. Existem muitas maneiras diferentes de começar e a quantidade de informação é tão grande que acabamos nos perdendo em meio a tantas possibilidades.

Ainda assim, quando vencemos esse primeiro obstáculo, escolhemos uma linguagem de programação e iniciamos o desenvolvimento, logo percebemos que precisamos escrever muito código para realizar tarefas essenciais como desenhar imagens na tela, emitir sons, capturar entrada através de gamepads, teclado ou mouse, entre outras. Com isso, acabamos perdendo muito tempo trabalhando para preparar toda essa base antes mesmo de começar a desenvolver a ideia do nosso jogo. É justamente nessa fase que a maioria dos aspirantes a desenvolvedores de jogos acaba desistindo.

Uma boa notícia é que muitos desenvolvedores já passaram por isso antes e decidiram resolver esse problema para facilitar o desenvolvimento de seus próximos jogos ou para ajudar outros desenvolvedores. Como essas tarefas são bastante comuns e como praticamente todos os jogos fazem uso delas, faz sentido escrever o código para todas elas uma única vez e disponibilizar esse código em forma de bibliotecas. Assim, utilizando essas bibliotecas podemos começar a codificar diretamente as regras do nosso jogo sem nos preocuparmos com as tarefas mais básicas.

Quando reunimos esse conjunto de bibliotecas com o objetivo de fornecer um *framework* para a execução de jogos, estamos criando o que chamamos de um ***game engine***. Um *game engine* (motor de jogos) é capaz de executar vários tipos de jogos diferentes assim como um motor pode ser usado para mover os mais diversos objetos como carros, esteiras, hélices e turbinas. Geralmente, um *game engine* possui um conjunto de subsistemas que são responsáveis por tipos específicos de tarefas. Os principais subsistemas de um *game engine* são apresentados abaixo:

- **Renderização 2D/3D** - responsável por todas as tarefas relacionadas ao desenho de objetos, efeitos especiais, vídeos na tela, entre outros.
- **Áudio** - responsável pela emissão de efeitos sonoros a partir de arquivos às vezes utilizando recursos mais avançados como sons posicionais 3D.
- **Física** - responsável pela simulação da movimentação dos objetos de um jogo com base nas leis da física como conhecemos.
- **Colisões** - responsável pela detecção de colisão entre os objetos de um jogo.
- **Scripts** - responsável pela interpretação de scripts contendo regras ou comportamento de objetos do jogo e que facilitam a customização por desenvolvedores, designers e artistas.
- **Entrada** - responsável pela captura dos comandos de dispositivos de entrada como teclado, mouse, gamepads e sensores diversos.

Hoje existem muitos *game engines* disponíveis para quem deseja desenvolver um jogo. Dentre os mais conhecidos podemos citar o **Unreal Engine** famoso pelas franquias **Unreal Tournament** e **Bioshock**; **Crytek Engine** conhecido pela série de jogos **Crysis**; **Source Engine** responsável por jogos de sucesso como **Half-life** e **Portal**; e o **Unity** mais famoso por jogos como **Hearthstone** e **Ori and the Blind Forest**.

Antigamente o uso desses *game engines* estava restrito a grandes produtoras devido ao custo elevado de aquisição das licenças mas recentemente tem havido uma mudança na estratégia de comercialização desses produtos. O **Unreal Engine 4**, o **Source Engine 2** e o **Unity 5** agora podem ser utilizados de graça sem a necessidade de se adquirir uma licença. Você só precisará adquirir uma licença ou pagar *royalties* depois de ter produzido e comercializado seu jogo.

O Unity

Apesar de termos várias opções de *engines* para desenvolvimento de jogos, o Unity vem ganhando bastante destaque nesse mercado bastante concorrido. Isso acontece porque o Unity foi criado desde o início tendo em mente a sua utilização tanto por desenvolvedores quanto por artistas e designers, que geralmente produzem recursos e documentação mas não são incluídos no processo de desenvolvimento de um jogo.

Isso acabou abrindo as portas para que eles pudessem contribuir mais ativamente durante o processo de desenvolvimento através de uma ferramenta integrada e expansível que permite que artistas possam realizar ajustes relativos ao visual de um jogo sem a necessidade de escrever uma linha de código sequer. O mesmo acontece com os designers que puderam auxiliar, por exemplo, no balanceamento dos jogos, na criação de novo conteúdo ou em ajustes simples das regras de jogo também sem precisar escrever código para isso.

Outra característica que permitiu a entrada do Unity nesse mercado tão concorrido foi a **Asset Store**. A Asset Store é uma loja de recursos que podem ser adquiridos (pagos ou grátis) e utilizados na criação de jogos. Esses recursos variam desde modelos 3D, imagens e sons até scripts, ferramentas adicionais e efeitos visuais. Qualquer usuário do Unity pode compartilhar ou vender seus recursos na Asset Store criando uma fonte bastante rica de recursos para qualquer um que queira criar um jogo.

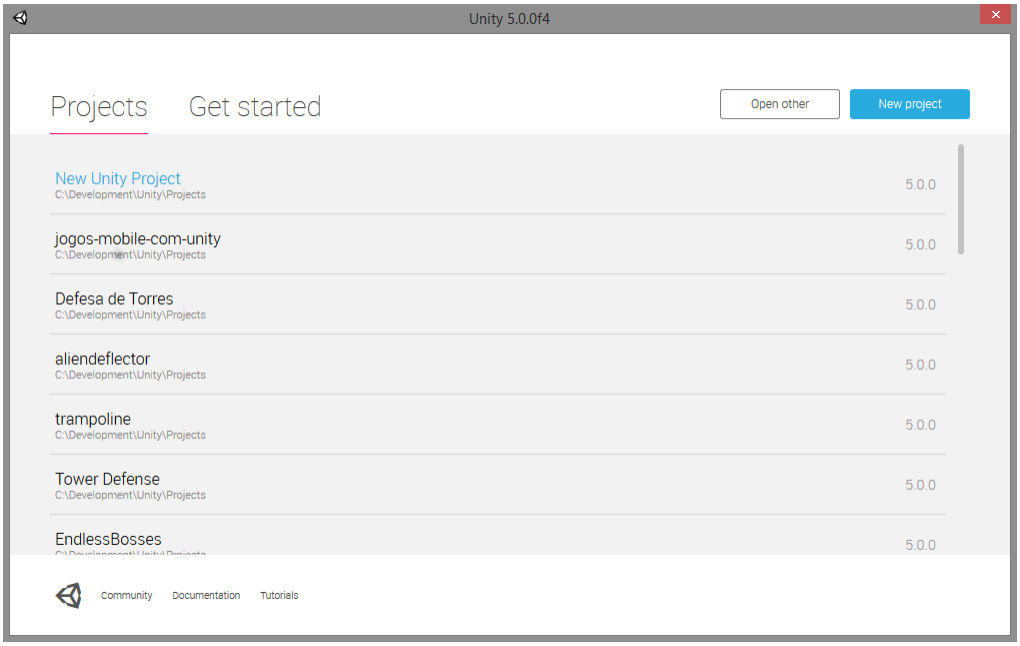
Uma outra grande vantagem do Unity é a sua capacidade de publicar um mesmo jogo em mais de 20 plataformas diferentes incluindo as plataformas Android, iPhone, Playstation (3, 4, Vita), XBox (360, One), Nintendo (Wii U, 3DS), PC Desktop, PC Web, entre outras.

Por todos esses motivos, escolhemos o Unity como a ferramenta mais adequada tanto para quem está começando no desenvolvimento de jogos quanto para quem já conhece um pouco do assunto e quer ganhar mais produtividade sem ter que reinventar a roda.

Criando um novo projeto

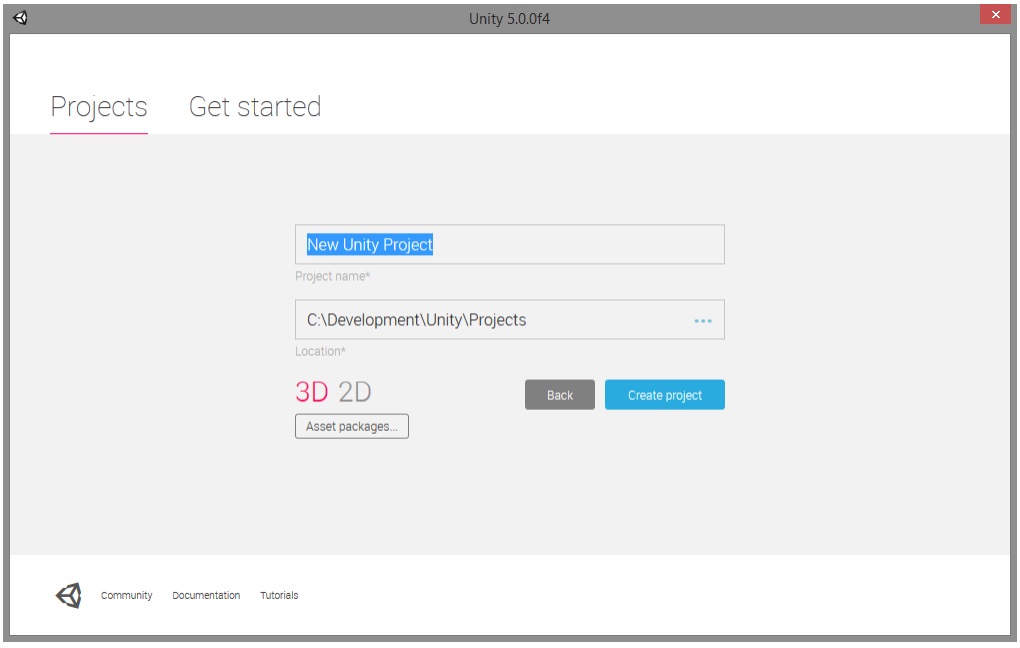
Antes de podermos criar um jogo no Unity precisamos baixar a última versão do *engine*. Para isso podemos visitar o site oficial do Unity: `http://www.unity3d.com` . Depois disso, basta visitar o link **Get Unity** e depois clicar no botão **Free Download** localizado ao final da coluna **Personal Edition**. Agora é só clicar no botão **Download Installer** para iniciar o download do instalador do Unity.

Depois de baixado o instalador do *engine*, basta executar o arquivo e seguir os passos do *wizard* de instalação. Com isso, um ícone do Unity vai ser criado no seu Desktop. Agora para iniciar o Unity, basta dar um duplo clique nesse ícone e aguardar a inicialização. A primeira tela exibida pelo Unity será a apresentada na figura abaixo:



Nesta tela o Unity mostra uma lista com os projetos mais recentes e também dois botões: **Open Other** prá abrir um projeto que não esteja na lista e **New Project** para criar um novo projeto.

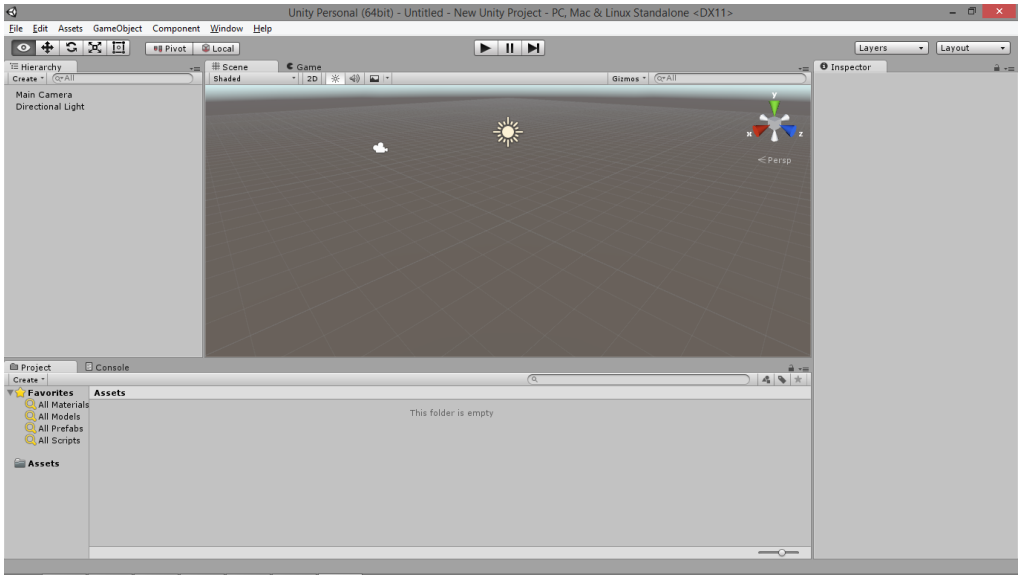
Para começarmos o desenvolvimento de um jogo no Unity precisamos criar um projeto então vamos utilizar o botão **New Project**. Depois de clicar nesse botão, o Unity mostrará a tela de criação de projetos mostrada abaixo:



Nesta tela podemos dar um nome para o nosso projeto em **Project Name** e escolher em que pasta ficarão os seus arquivos em **Location**. Podemos também dizer se o nosso projeto será um jogo **2D** ou **3D** clicando nos textos de mesmo nome nessa janela. Depois de preenchidos os campos, podemos criar em **Create Project** para criar o projeto e abrir o editor do Unity pronto prá começar!

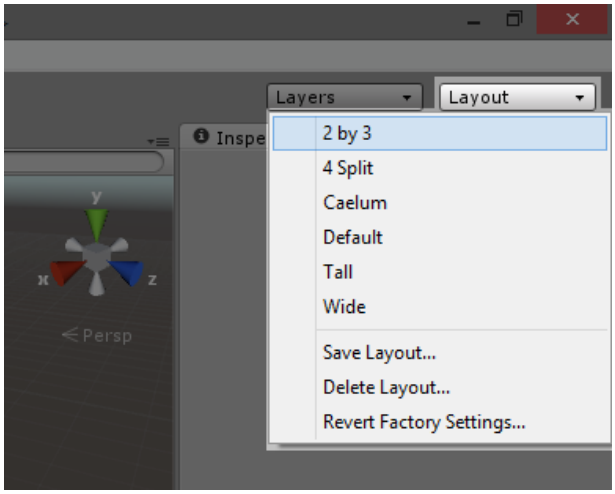
O editor do Unity

Sempre que abrirmos um projeto no Unity, a primeira tela que veremos será a janela do editor apresentada abaixo:

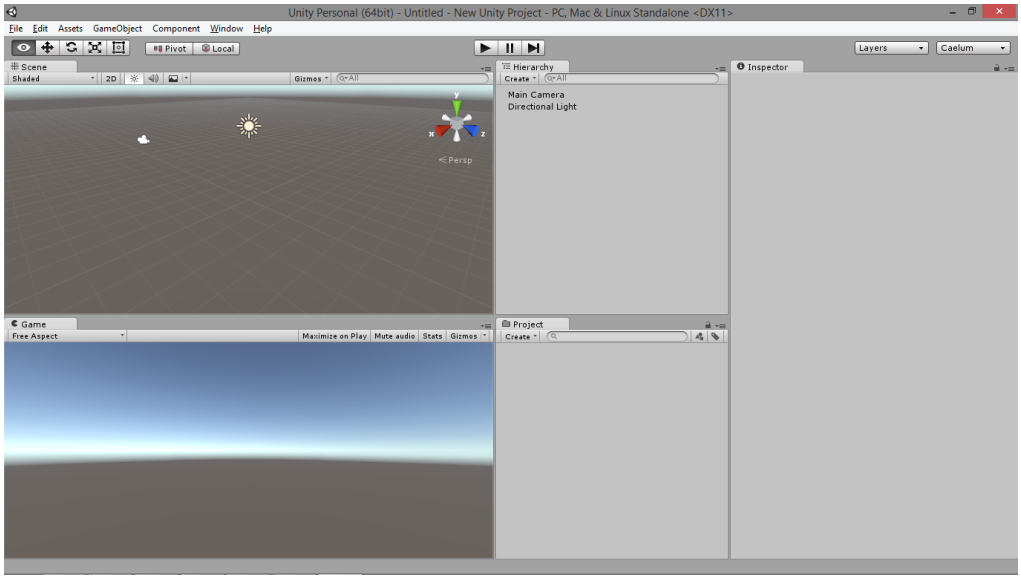


É no editor que passaremos a maior parte do tempo durante a criação do nosso jogo. Por esse motivo é interessante configurar o layout do editor para que as ferramentas mais importantes fiquem sempre ao nosso alcance.

O editor do Unity já possui algum padrões de layout de tela pré-configurados para o uso. Para acessá-los devemos clicar na caixa de opções de nome **Layout** logo acima da aba **Inspector**, localizada do lado direito da janela como ilustrado na figura abaixo:



Apesar de já possuir algum padrões prontos, recomendamos começar com o padrão **2 by 3** e alterar a posição de algumas abas para facilitar o trabalho dentro do editor até alcançar a configuração mostrada abaixo:



Nesta configuração, temos do lado esquerdo as abas **Scene** e **Game**, que são visualizações do nosso jogo, e do lado direito as abas **Hierarchy**, **Project** e **Inspector**, que serão usadas para organização, criação e configuração dos objetos e recursos do jogo.

Com isso estamos prontos para começar a criação do nosso jogo!

As regras do jogo: Defesa de Torres

Durante o curso construiremos um jogo do início ao fim começando pela criação do projeto e adicionando aos poucos novas funcionalidades até ter um jogo completo.

O jogo que criaremos será um *tower defense*. Neste jogo o jogador deve construir torres em posições estratégicas para criar uma linha de defesa contra inimigos invasores. Caso um inimigo consiga cruzar a linha de defesa sem ser destruído, o jogador é penalizado com a perda de uma vida. Se as vidas do jogador acabarem então é fim de jogo. O objetivo final é sobreviver o maior tempo possível à invasão.

Construindo o cenário do jogo com Game Objects

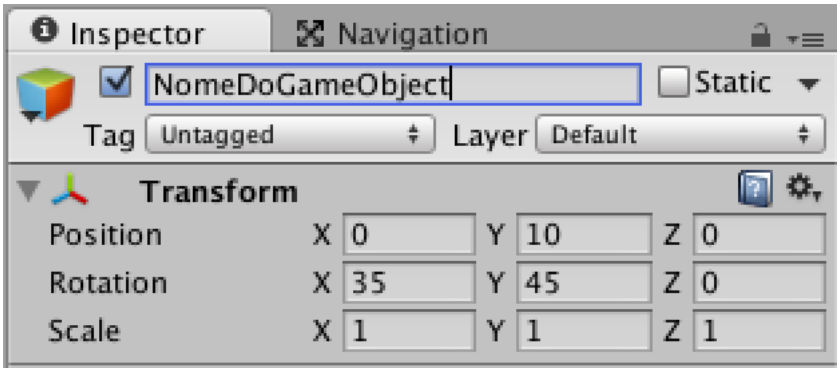
O que são Game Objects?

No Unity, cada objeto dentro de uma cena é chamado de ***Game Object***, por exemplo: personagens, luz, câmera, partículas... Se todos esses diferentes objetos são *Game Objects*, como diferenciamos uma câmera de uma partícula?

Por padrão, um *Game Object* não tem nenhum comportamento específico. Sua diferenciação é feita a partir do conjunto de componentes (*components*) que escolhemos para esse objeto. Então, uma câmera é um *Game Object* que possui um conjunto de componentes diferentes de uma partícula, que também é um *Game Object*!

Como todo *Game Object* é um objeto dentro da nossa cena, como fazemos para posicioná-lo num ponto específico do nosso ambiente?

Uma componente muito importante dos *Game Objects* é o **Transform**, que permite configurarmos vários atributos interessantes, como escala (*scale*), rotação (*rotation*) e **posição** (*position*).



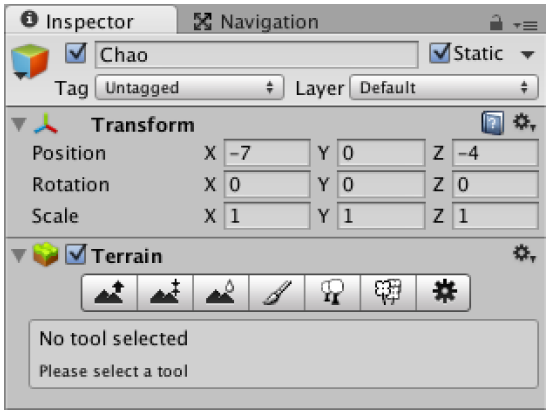
Então, para configurarmos a posição de qualquer *Game Object* na nossa cena, basta acessarmos o atributo `Position` do componente `Transform` !

Terreno: nosso segundo Game Object

Temos nosso inimigo criado, mas ele ainda está "flutuando" na nossa cena. Para melhorar isso, vamos criar o chão do nosso jogo!

Como o chão é um objeto da nossa cena, criaremos outro *Game Object*: o ***Terrain***.

Ao clicarmos na aba `Hierarchy -> Create -> 3D Object -> Terrain` , veja que esse *Game Object* criado possui um outro componente além do `Transform` , chamado `Terrain` :



Nesse componente podemos customizar as propriedades do nosso chão, como largura, profundidade, resolução, cores... Além disso podemos criar elevações e depressões no relevo desse terreno!

Manipulando a Câmera

Logo ao criarmos nosso projeto no Unity, a própria ferramenta já cria uma cena com dois *Game Objects*: uma câmera principal e uma luz padrão. Podemos alterar seus atributos!

Podemos clicar na câmera principal (`Main Camera`) e, na aba `Inspector` , manipular os seus componentes, como o `Transform` e `Camera` .

No componente `Camera` , podemos manipular o atributo `Field of View` que representa o ângulo de visão da nossa câmera.

Para saber mais: Field of View

O olho humano possui um ângulo de visão (_field of view_) de aproximadamente 180 graus, porém num jogo em primeira-pessoa costumamos usar valores bem mais baixos que esse pois o cérebro humano acredita estar olhando para uma simples

janela.

```
É bastante comum encontrarmos jogos com _field of view_ em torno de
70 graus, aumentando de acordo com a quantidade de monitores ligados até 120
graus.
```

Movimentando o inimigo por um caminho

Dando vida ao jogo

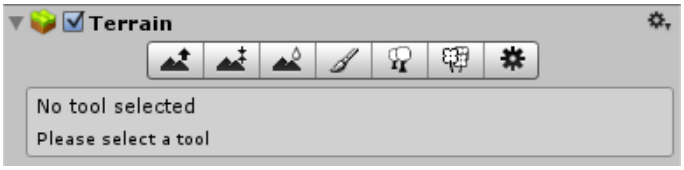
Agora que já temos uma cena com alguns dos principais elementos do nosso jogo, vamos começar a adicionar algumas funcionalidades começando pelo movimentação do inimigo.

Da forma como o jogo se encontra no momento, nosso inimigo poderia se movimentar em qualquer direção sem nenhum obstáculo em seu caminho. Além disso, dessa forma não ficaria claro para o jogador onde o inimigo está querendo chegar.

Para melhorar isso, vamos começar adicionando detalhes ao nosso terreno para mostrar qual será o caminho a ser percorrido pelo inimigo. Para isso vamos precisar fazer modificações no objeto `Chao` que criamos anteriormente.

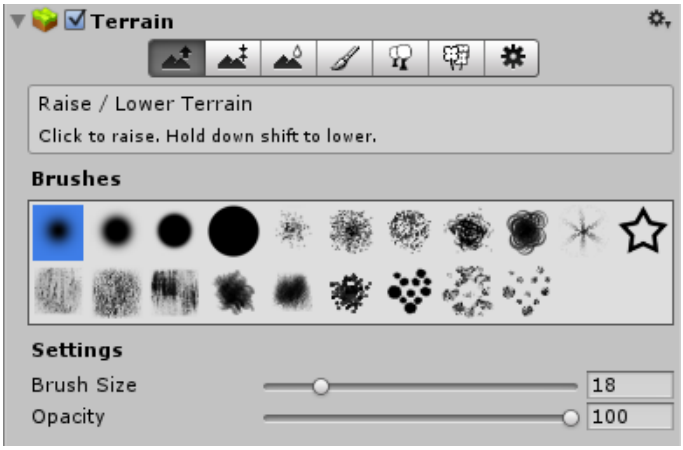
Terrain e suas ferramentas

O objeto `Chao` é um `Terrain` que permite a modelagem de terrenos de forma bem simples e intuitiva. Quando selecionamos um objeto do tipo `Terrain` , o `Inspector` exibe o componente `Terrain` que utilizaremos para desenhar o terreno do nosso jogo. Inicialmente, nesse componente temos apenas uma barra de ferramentas como na figura abaixo:



A primeira ferramenta que vamos usar é a `Raise / Lower Terrain` correspondente ao primeiro botão da barra de ferramentas. Essa ferramenta serve para definir o relevo do nosso terreno permitindo a criação de elevações e depressões de forma bem fácil!

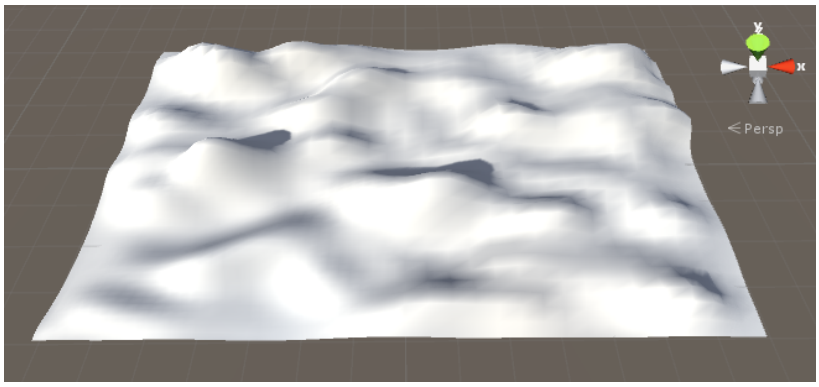
Ao clicar nessa ferramenta, o componente irá exibir no `Inspector` as suas opções como ilustrado na figura:



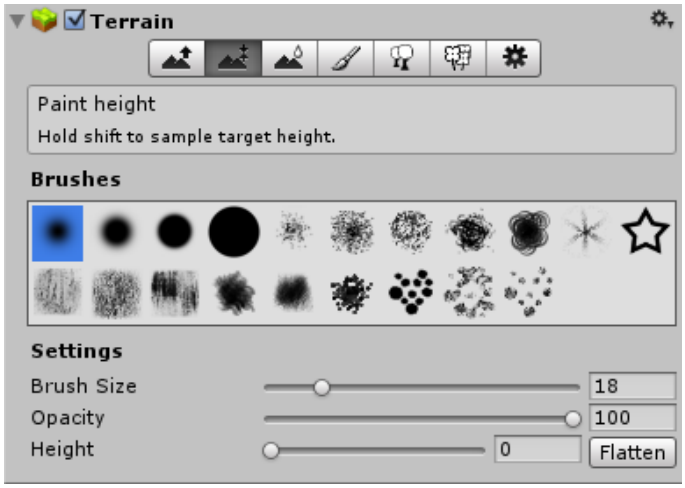
Na seção `Brushes`, podemos escolher o formato do pincel que usaremos para criar elevações ou depressões no terreno. Logo abaixo, temos a seção `Settings` onde podemos escolher o tamanho do pincel em `Brush Size` e a sua opacidade (força) em `Opacity`.

Após configurar essas opções, podemos criar elevações no terreno clicando e arrastando o mouse sobre o terreno na aba `Scene`. Para diminuir a elevação do terreno e criar depressões, basta clicar e arrastar sobre o terreno segurando a tecla **SHIFT**.

Agora que já sabemos como usar a ferramenta, podemos criar um terreno irregular bem mais interessante que o anterior! Por exemplo, podemos modelar um terreno como o abaixo:



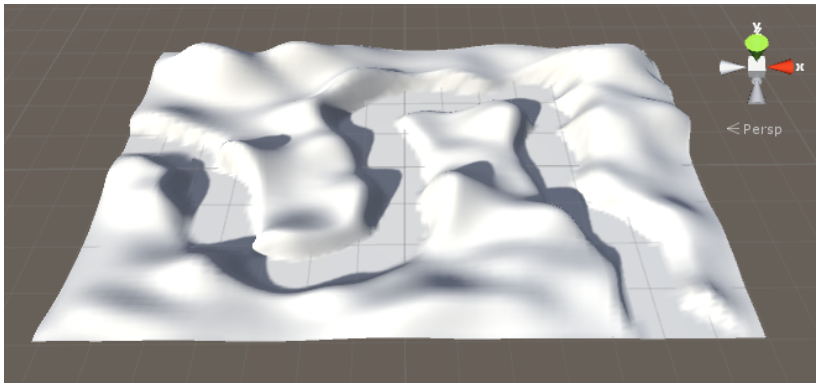
Para delimitar o caminho, vamos utilizar mais uma ferramenta do componente `Terrain`. Novamente, com o `Chao` selecionado, vamos agora utilizar a segunda ferramenta desse componente mostrada na figura abaixo:



Essa ferramenta serve para criar regiões que possuam uma mesma altura escolhida por nós. Dessa forma, fica fácil definir áreas planas no terreno.

Assim como na ferramenta anterior, temos uma seção `Brush` e uma seção `Settings` com as mesmas funcionalidades. A diferença está no funcionamento da ferramenta e na adição do atributo `Height` e do botão `Flatten`. Para utilizar a ferramenta basta clicar e arrastar sobre o terreno na aba `Scene`. Nesse caso, a ferramenta irá modificar a altura do terreno na região clicada para a altura definida no atributo `Height`. Além disso, também podemos resetar a altura do terreno inteiro de uma só vez definindo a altura em `Height` e clicando no botão `Flatten`.

Então agora podemos delimitar o caminho a ser percorrido definindo `Height` como `0` e desenhando um caminho na aba `Scene` começando no lado esquerdo do terreno e terminando no lado direito. Por exemplo, podemos desenhar um caminho como o exibido na figura abaixo:



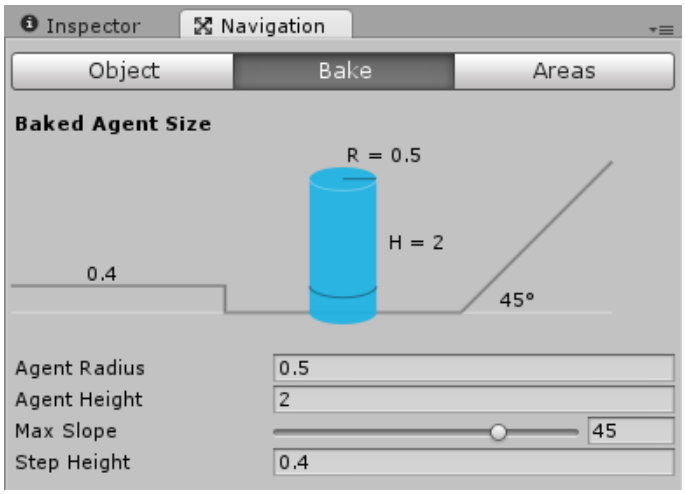
Como agora definimos um contraste entre o caminho e o restante do cenário, fica fácil para o jogador perceber por onde o inimigo irá passar e onde ele pretende chegar.

Delimitando a área de movimento com a NavMesh

Com o caminho definido, precisamos agora fazer com o que inimigo consiga se mover por este caminho sem se desviar. Mas como vamos fazer nosso inimigo saber por onde ele pode andar? Para isso utilizaremos o que chamamos no Unity de `NavMesh` .

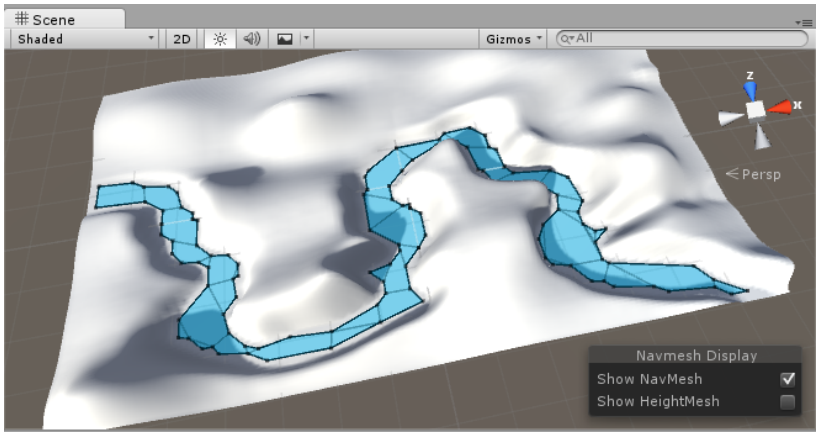
Uma `NavMesh` (abreviação de *navigation mesh*) é um modelo poligonal que serve para indicar por onde os objetos podem se mover. Além disso, a `NavMesh` permite que os objetos encontrem facilmente o menor caminho para se chegar em um determinado ponto.

Para criar uma `NavMesh` , vamos abrir a janela `Navigation` indo no menu `Window -> Navigation` . Antes de gerar a `NavMesh` precisamos configurar alguns parâmetros para indicar onde nosso inimigo pode caminhar. Na janela `Navigation` , vamos clicar no botão `Bake` para ter acesso às opções abaixo:



Esses parâmetros são utilizados para dizer ao Unity como a `NavMesh` deve ser gerada. Os atributos `Agent Radius` e `Agent Size` servem para indicar qual será o tamanho dos objetos que se moverão pela `NavMesh` . O atributo `Max Slope` indica qual a inclinação máxima pela qual um objeto pode se mover e o atributo `Step Height` qual a altura máxima aceitável para um degrau que o objeto pode subir ou descer.

Com os parâmetros configurados, basta clicar no botão `Bake` na parte inferior da janela `Navigation` . O Unity irá gerar a `NavMesh` e apresentá-la na aba `Scene` como na figura abaixo:

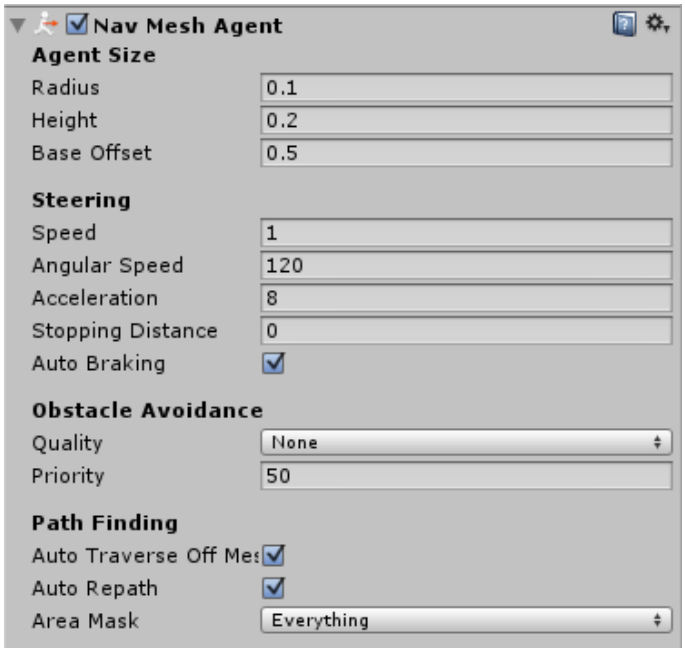


A `NavMesh`, apresentada como uma malha na cor azul sobre o terreno, representa a área que poderá ser acessada por um objeto que se mova sobre ela. Caso a `NavMesh` apresente alguma região muito estreita ou não forme uma única área contínua, os objetos não conseguirão se locomover corretamente. Nesses casos, precisaremos gerar novamente a `NavMesh` com novos parâmetros ou depois de fazer ajustes no terreno.

Transformando o inimigo em um NavMeshAgent

Agora que criamos a `NavMesh`, como fazemos para dizer para o nosso inimigo que ele deve se mover respeitando os limites dela?

A forma de fazer isso no Unity é indicando que o inimigo é um objeto que sabe interagir com uma `NavMesh`. Para isso, basta adicionar um componente chamado `NavMeshAgent` ao nosso objeto `Inimigo`. Na ferramenta, selecionamos o objeto `Inimigo` e no `Inspector` clicamos em `Add Component -> Navigation -> Nav Mesh Agent`. Agora no `Inspector`, temos várias novas opções relacionadas ao movimento desse objeto na `NavMesh` como apresentado na figura abaixo:



Para adequar o nosso inimigo ao `NavMesh`, precisamos indicar o tamanho do nosso objeto alterando seu raio em `Radius` e sua altura `Height` em `Agent Size`. Além disso, alteramos também a velocidade do objeto `Speed = 1` em `Steering`. Para finalizar, alteramos o atributo `Quality = None` em `Obstacle Avoidance` para que o inimigo possa se locomover sem se preocupar em colidir com outros objetos ou com o cenário.

Fazendo o inimigo se mover com script

Tendo criado a `NavMesh` e transformado o inimigo em um `NavMeshAgent`, como fazer para dizer ao inimigo para onde ele deve ir?

Para fazer isso vamos ter que criar um novo comportamento para o nosso inimigo dizendo que ele deve se locomover para o fim do caminho assim que o jogo começar.

Comportamentos de objetos no Unity são definidos através de scripts escritos em C#. Todo script depois de criado poderá ser usado como um componente em qualquer objeto do Unity. Para criar um novo script, podemos selecionar o objeto `Inimigo` e depois clicar em `Inspector -> Add Component -> New Script`. Depois basta preencher o campo `Name = Inimigo` e clicar em `Create and Add`.

Para editar o script, podemos dar um duplo clique em `Project -> Inimigo`. Neste momento, o Unity abrirá uma ferramenta externa para edição de scripts: o **MonoDevelop**.

No MonoDevelop, o script `Inimigo` será exibido contendo o seguinte código:

```
using UnityEngine;
using System.Collections;
using UnityEngine.AI;

public class Inimigo : MonoBehaviour
{

    // Use this for initialization
    void Start ()
    {

    }

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

    }
}
```

Note que o script `Inimigo` herda da classe `MonoBehaviour` para indicar que o nosso script é um comportamento do Unity e como tal poderá ser utilizado como um componente no editor. Além disso, temos os métodos `Start` e `Update`. Por enquanto, utilizaremos apenas o `Start` que é o método invocado automaticamente pelo Unity **no momento que o objeto é criado**.

Vamos aproveitar esse método para ordenar que o inimigo se mova até o fim do caminho utilizando a `NavMesh` como guia. Mas quem é o componente responsável por fazer com que nosso objeto interaja com a `NavMesh`? O `NavMeshAgent`! Então vamos ter que pedir para o `NavMeshAgent` realizar essa tarefa.

Para manipularmos algum componente de um objeto no script, precisamos ter uma referência para ele. Conseguimos essa referência utilizando o método `GetComponent` como no código abaixo:

```
NavMeshAgent navMeshAgent = GetComponent<NavMeshAgent>();
```

Agora que temos uma referência para o componente `NavMeshAgent` do nosso `Inimigo`, basta invocar o método `SetDestination` neste componente como no código abaixo:

```
public class Inimigo : MonoBehaviour
{
  void Start ()
  {
    NavMeshAgent agente = GetComponent<NavMeshAgent>();
    agente.SetDestination (/*como saberemos a coordenada do fim do caminho?*/);
  }
}
```

Mas ainda temos um problema: o método `SetDestination` recebe uma posição como parâmetro indicando prá onde o objeto deve se mover. Como vamos indicar a posição do final do caminho?

Para não termos que especificar as coordenadas do final do caminho na mão, podemos criar um *game object* vazio com o nome `FimDoCaminho` em nossa cena e posicioná-lo no fim do nosso caminho. Assim podemos utilizar a posição desse objeto como referência de posição para o método `SetDestination`!

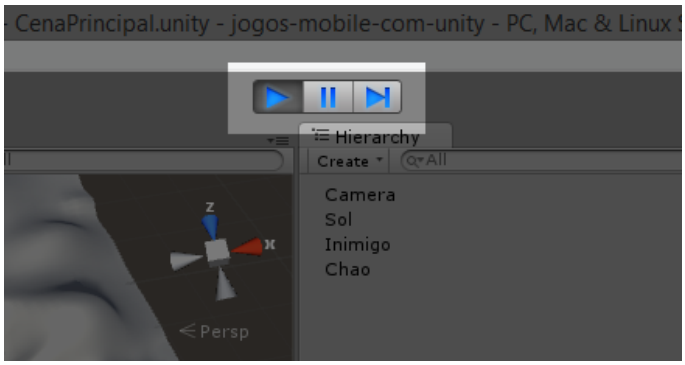
Ainda assim, precisamos de uma forma de obter uma referência para esse objeto. Nesse caso, podemos utilizar o método `GameObject.Find` passando o nome de um objeto da cena como parâmetro para receber uma referência para esse objeto:

```
GameObject fimDoCaminho = GameObject.Find ("FimDoCaminho");
```

Finalmente, tendo a referência para o objeto, podemos acessar o atributo `transform.position` desse objeto para conseguir sua posição. Juntando tudo isso, teremos o seguinte código no script `Inimigo`:

```
public class Inimigo : MonoBehaviour
{
    void Start ()
    {
        NavMeshAgent agente = GetComponent<NavMeshAgent>();
        GameObject fimDoCaminho = GameObject.Find ("FimDoCaminho");
        Vector3 posicaoDoFimDoCaminho = fimDoCaminho.transform.position;
        agente.SetDestination (posicaoDoFimDoCaminho);
    }
}
```

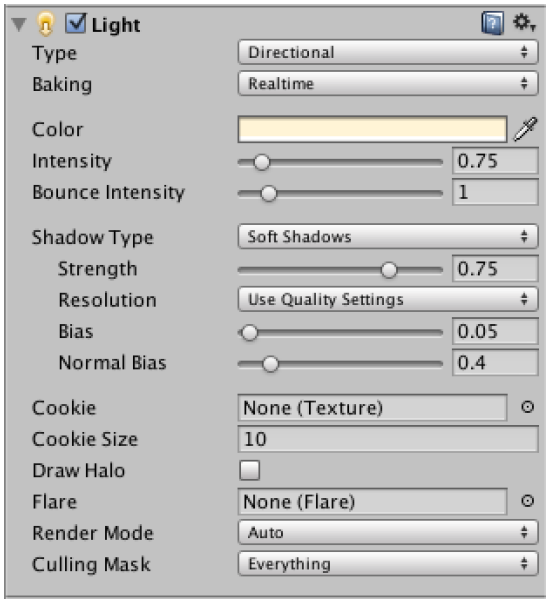
Com o script finalizado, basta rodar o jogo agora para verificar que tudo está funcionando. Vamos então clicar no botão de `Play` localizado na barra de ferramentas superior do editor como indicado na imagem abaixo:



Quando estamos executando o jogo, o editor do Unity entra no *play mode*. Durante o teste do jogo, podemos manipular nossa cena criando ou removendo objetos e também alterando seus parâmetros através do `Inspector`. Isso é ótimo para fazer um ajuste fino de detalhes que são melhores vistos enquanto o jogo está rodando. Mas fique alerta! Todas alterações realizadas no *play mode* são desfeitas quando encerramos a sessão de teste clicando novamente no botão `Play`.

Luz ambiente do nosso jogo

Além da câmera padrão, temos uma *directional light* criada por padrão pronta para manipularmos no nosso jogo! Dessa vez, podemos alterar o componente `Light`.



Além da sua cor (*Color*), podemos configurar a intensidade (*Intensity*) da luz, sua sombra (*Shadow type*) e até mesmo sua reflexão nos objetos (*Bounce Intensity*).

Veja que temos quatro tipos de luz:

- **Point:** esse tipo de luz tem o máximo de iluminação no seu centro e vai escurecendo quanto mais longe estamos desse centro. É usado quando queremos simular um poste de luz, ou uma lâmpada, ou até mesmo explosões.
- **Spot:** é bem parecido com o *point light*, mas fica confinada a um cone de direção. Faróis de carros, luzes de palco são bons exemplos desse tipo de luz.
- **Directional:** neste tipo de luz não temos uma origem definida, nos importa somente a sua direção. Podemos simular a luz do sol ou da lua com ela.
- **Area:** em vez de criarmos vários *point lights*, podemos criar uma iluminação numa área. Neste caso os raios de luz serão disparados em várias direções dentro desta área. Dessa forma podemos simular uma iluminação dentro de casa de forma mais realista.

Como o cálculo de iluminação é algo bastante custoso para o processador, podemos determinar se queremos uma iluminação em tempo-real ou pré-calculada pela *engine* alterando a opção `Baking`.

Porém, podemos fazer essa configuração para todas as luzes da nossa cena acessando um painel específico de iluminação chamado **Lighting**.

Neste painel, podemos desmarcar a opção `Continuous Baking`.