https://github.com/AnomalyGameStudio/tutoriais/wiki/Boas-Pr%C3%A1ticas-da-Unity-3D

Unity3D Best Practices

Game Object vazios (empty game objects) nomeados devem ser usados como pastas nas Scenes. Scenes que são organizadas com cuidado facilitam a procura de Objects.

Coloque os Prefabs e as pastas (Empty Game Object) nas coordenadas 0 0 0. Se a Transform não é especificamente usada no posicionamento de um Object, esse deve estar na origem (x= 0, y= 0, z= 0). Deste modo tende-se a ter menos perigo de encontrar problemas com posicionamento local e global, tornando o código mais simples.

Coloque o chão do World na coordenada y = 0. Isso facilitara na hora de colocar objetos no chão, e o World poderá ser tratado como 2D (quando apropriado) na lógica, na inteligência artificial e física do jogo.

Crie todas as Meshes olhando para a mesma direção (eixo Z positivo ou negativo). Isso se aplica para todas as Meshes que tenham o conceito de direção, por exemplo, personagens. Muitos algoritmos serão simplificados se todos encararem a mesma direção originalmente.

Use a escala certa desde o princípio. Quando criar as artes, faca de forma com que elas possam ser diretamente importadas sem precisar modificar a escala (scale transform x=1, y=1, z=1). Use o cubo padrão do Unity para fazer comparações de escala. Escolha uma razão entre a escala global para unidade do Unity, e guarde-a durante o resto do desenvolvimento.

Use Prefabs para tudo. Os únicos objetos na Scene que não são Prefabs devem ser pastas(Empty Game Object). Até mesmo objetos únicos que devem ser *usados somente uma vez* devem ser Prefabs. Isso facilita na hora de fazer modificações que não necessitam mudanças na Scenes.

Use prefabs separados para especialização; não especialize instâncias. Se você tiver dois tipos de inimigos e eles apenas se diferenciam pelas

propriedades, faça prefabs separados para as propriedades e ligue a eles. Isto deixa possível:

- fazer mudanças para cada tipo em um mesmo lugar
- fazer mudanças sem ter que mudar a scene.

Se você tiver muitos tipos de inimigos, a especialização ainda assim não devem ser feitas na instância no editor. Uma alternativa é fazer processualmente, ou usando um arquivo central ou prefab para todos os inimigos. Um simples campo dropdown pode ser usado para diferenciar os inimigos, ou um algoritmo baseado na posição do inimigo ou o progresso do player.

Não coloque os meshes como raiz dos prefabs se você vai adicionar outros scripts. Quando você faz um prefab de um mesh, primeiro coloque o mesh em um *Game Object* vazio e faça esse ser a raiz. Coloque os scripts na raiz, não no nó do mesh. Deste jeito é muito mais fácil trocar o mesh por outro mesh sem perder os valores que você marcou no inspector.

Mantenha sua própria classe Time para realizar pausas mais facilmente. Utilize Time.DeltaTime e Time.TimeSinceLevelLoad para 'contabilizar' pausas e TimeScale. Isto exigirá disciplina dos programadores, mas tornará as coisas mais fáceis e organizadas em longo prazo, especialmente quando executado de locais diferentes (Ex: animações de interfaces, e animações de jogo).

Use *singletons* por conveniencia. A classe a seguir fará de qualquer classe herdada um *singleton*automaticamente:

```
public class Singleton<T> : MonoBehaviour where T : MonoBehaviour
{
   protected static T instance;

   /**
      Returns the instance of this singleton.
   */
   public static T Instance
   {
      get
      {
         if(instance == null)
         {
            instance = (T) FindObjectOfType(typeof(T));

            if (instance == null)
            {
               Debug.LogError("An instance of " + typeof(T) +
                  " is needed in the scene, but there is none.");
```

```
            }
        }

        return instance;
    }
  }
}
```

Singletons são úteis para *managers*,
como ***ParticleManager*** or ***AudioManager*** or ***GUIManager***.

- Evite usar *singletons* para instâncias únicas de *prefabs* que não são *managers* (como *Player*). Não aderir a este princípio complica hierarquias de herança, e torna certos tipos de mudança mais difíceis. Ao invés disso, mantenha referências a elas no seu *GameManager* (ou outra classe Divina apropriada ;-))
- Defina propriedades e métodos estáticos para variáveis públicas e métodos que são utilizados frequentemente de fora da classe. Isso permite que você escreva *GameManager.Player* ao invés de *GameManager.Instance.player*

Para componentes, nunca faça variáveis públicas que não devam ser ajustadas no *inspector* **. Do contrário, elas *serão* alteradas pelo *designer*, especialmente se não for claro o que ela faz. Se parecer inevitável, use a *tag* **[HideInInspector]. Não torne as coisas públicas para que elas apareçam no inspector. Apenas propriedades ou membros que você quer alterar de fora de sua classe devem ser públicos. Ao invés disso, utilize o atributo *[SerializeField] (@SerializeField)* para tornar variáveis visíveis no inspector. Se você tem alguma coisa que necessita ser pública mas não editável pelo designer, então utilize [HideInInspector] (@HideInInspector). Ainda sobre isso, não torne coisas públicas apenas por diversão. Quanto mais funções públicas e variáveis membro você tiver em uma classe, mais coisas são exibidas na lista *drop-down* quando você acessá-la. Deixe tudo limpo.

Não use strings para qualquer coisa além de mostrar texto. Em particular, não use strings para identificar objetos ou prefabs etc. Uma infeliz exceção são as animações, que geralmente são acessadas com seus nomes em *string*.

Tag/Layer: Quando checar uma tag, use o método *CompareTag* no componente ou *GameObject*.

Se você tem muita história em texto, coloque-os em um arquivo. Não coloque-os em um campo para edição no inspector. Faça que seja fácil de alterar sem a necessidade de abrir o editor do Unity, e especialmente sem que seja necessário salvar a Scene.

Se você planeja traduzir, separe todas as strings em um local. Existem muitas formas de fazer isso. Uma maneira é definir um Text Class com um campo string público para cada string, com o padrão marcado como Inglês, por exemplo. Outras linguagens herdam essa classe e reescrevem os campos com suas próprias traduções equivalentes.

Técnicas mais sofisticadas (apropriada quando o campo de texto é muito grande e/ou o numero de linguagens é grande) vão ler uma planilha e entender qual é a string certa baseada na linguagem escolhida.

Faça seu próprio contador de FPS. Sim. Ninguém sabe o que esse contador de FPS do Unity realmente mede, mas não é frame rate (frequência dos quadros). Programe o seu próprio assim o número pode corresponder com a intuição e inspeção visual.

Siga uma convenção de nomeação e estrutura de pastas documentada. Nomeação e estrutura de pastas consistentes deixam fácil de achar as coisas e imaginar o que são elas.

###Princípios Gerais de Nomeação###

1. Chame a coisa pelo que ela é. Um pássaro deve ser chamado de Pássaro.
2. Escolha nomes que podem ser pronunciados e lembrados. Se fizer um jogo sobre os Mayas não chame seu level de QuetzalcoatisReturn.
3. Seja consistente. Quando escolher um nome, use-o.
4. Use PascalCase como este: ComplicatedVerySpecificObject. Não use espaços, underlines ou hífens, sem exceções (veja a área de Nomeando Aspectos Diferentes de uma Mesma Coisa).
5. Não use números de versões ou palavras para indicar seu progresso (WIP, Final).
6. Não use abreviações: DVamp@W deve ser DarkVampire@Walk .

7. Use a terminologia descrita no documento de design (GDD): se no documento chamar a destruição de uma animação de *Die*, então use DarkVampire@Die e não DarkVampire@Death .
8. Mantenha o descritor mais especifico à esquerda: use DarkVampire e não VampireDark; PauseButton e não ButtonPaused. Deixando, por exemplo, mais fácil de achar o botão pause no *inspector* se todos os botões não começarem com a palavra *Button* (Muitas pessoas preferem o inverso, porque isso torna o agrupamento visual mais óbvio. No entanto, nomes não são para agrupamento, pastas sim. Nomes são para distinguir os objetos do mesmo tipo para que sejam encontrados de forma confiável e rápida).
9. Alguns nomes formam uma sequência. Use números nesses nomes, por exemplo: PathNode0.PathNode1. Sempre comece com 0 e não 1.
10. Não use os números em coisas que não formam uma sequência. Por exemplo: Bird0, Bird1 e Bird2 devem ser Flamingo, Eagle e Swallow.
11. Coloque dois underlines como um prefixo em objetos temporários: __Player_Backup.

###Nomeando Aspectos Diferentes de uma Mesma Coisa###

Use underlines entre o nome principal e a palavra que descreve o "aspecto". Por exemplo:

- **Estados dos botões GUI** EnterButton_Active, EnterButton_Inactive
- **Texturas** DarkVampire_Diffuse, DarkVampire_Normalmap
- **Skybox** JungleSky_Top, JungleSky_North
- **Grupos de LOD** DarkVampire_LOD0, DarkVampire_LOD1

Não use esta convenção apenas para distinguir entre tipos diferentes de itens, por exemplo Rock_Small, Rock_Large devem ser SmallRock, LargeRock.

###Estrutura###

A organização das Scenes, pastas do projeto e pastas dos scripts devem seguir um padrão similar.

####Estrutura de Pastas####

```
Materials
GUI
Effects
Meshes
   Actors
      DarkVampire
      LightVampire
      ...
```

```
   Structures
      Buildings
      ...
   Props
      Plants
      ...
   ...
Plugins
Prefabs
   Actors
   Items
   ...
Resources
   Actors
   Items
   ...
Scenes
   GUI
   Levels
   TestScenes
Scripts
Textures
GUI
Effects
...
```

####Estrutura da Scene####

```
Cameras
Dynamic Objects
Gameplay
   Actors
   Items
   ...
GUI
   HUD
   PauseMenu
   ...
Management
Lights
World
   Ground
   Props
   Structure
   ...
```

####Estrutura das pastas de Script####

```
ThirdParty
   ...
MyGenericScripts
   Debug
   Extensions
   Framework
   Graphics
   IO
   Math
   ...
MyGameScripts
   Debug
   Gameplay
```

```
    Actors
    Items
    ...
  Framework
  Graphics
  GUI
  ...
```

Não use Enum.ToString() para converter de um Enum para uma string. Enums não são muito confiáveis em questão de velocidade quando se faz isso. Você deve, em vez disso, usar Enum.GetName(typeof(AttackType), AttackType.Melee); Os testes mostram que Enum.GetName é duas vezes mais rápida do que toString.

Não use GameObject.Find para encontrar algo, em vez disso, faça-o como um SerializeField e arraste-o lá se possível. Seus designers vão te amar por isso no final, mesmo sendo chato de configurar as vezes.

A classe Transform herda de IEnumerable, o que significa que voce pode facilmente iterar pelos filhos em um foreach. Apenas lembre que sentenças de foreach declaram uma variável que é adicionada à memória e essa memória não será esvaziada no IOS, então isto causa falha de memória.

Evite strings. Strings são imutavéis no .NET e alocadas na memória. Você não pode as manipular como na linguagem C. Para UI's, use o StringBuilder para construir strings de uma maneira mais eficiente. Atrase a conversão para string até o mais tarde possível. Você pode usá-las como chaves, por que os literais devem apontar para a mesma instância na memória, mas não as manipule muito.