Projeto Coma
Mecânica Aponte e Clique
Macaco

Manual do Programador

Jonas Luz Jr.

Versão Alpha 1 (WIP)

Universidade de Fortaleza
Especialização em Animação e Jogos Eletrônicos
turma 1

Projeto COMA
Mecânica Aponte e Clique
MAC~CO - versão Alpha 1 (WIP)

Manual do Programador

Jonas de Araújo Luz Jr.

Fortaleza, Janeiro de 2014

Sumário

O Projeto Coma

O Projeto Coma tem por objetivo o desenvolvimento de um jogo eletrônico para fins de aprovação nas disciplinas do Prof. Daniel Siqueira, do curso de Especialização em Animação e Jogos Eletrônicos, turma 1 (2012-2013), da Universidade de Fortaleza (Unifor). Toda a turma foi envolvida, tendo sido dividida em subequipes especializadas.

O histórico do projeto pode ser acompanhado através do conjunto de links mantido sob o endereço http://abre.la/ProjetoComa

A Mecânica Aponte e Clique ~ Coma (MAC~CO)

Este manual documenta o trabalho realizado pelo estudante **Jonas de Araújo Luz Jr.**, que ficou responsável pela implementação do código da mecânica "aponte e clique" (*point'n'click*) a ser utilizada em todas as fases do jogo. Para atender a este requisito fundamental, a mecânica foi implementada sob a forma de uma *engine* reutilizável, batizada de MAC~CO (pronuncia-se "macaco"), um conveniente acrônimo de "mecânica aponte e clique ~ Coma".

A presente versão da MAC~CO é a alpha 1, utilizada na versão de mesma nomenclatura do próprio jogo Coma. Nesta edição, a mecânica possui as funcionalidades necessárias para implementação da fase 1 do jogo, que foi utilizada para homologar a *engine*, o que é pouco. Sabe-se que, para se tornar uma engine "aponte e clique" completa ainda há muito a fazer. Registre-se, portanto, que o resultado presente é ainda um trabalho em andamento, ou, em inglês, *work in progress* (WIP).

Quadro Técnico

Nome completo:	Mecânica Aponte e Clique ~ Coma
Nome resumido (acrônimo):	MAC~CO
Plataforma (*game engine*):	Unity 3D
Linguagem:	JavaScript (UnityScript)
Versão:	Alpha 1
Website:	http://mac-co.jonasluz.com.br

Licenciamento

A implementação da *engine* MAC~CO é um trabalho acadêmico. Seu principal objetivo é o estudo e o aprendizado. Como tal, é natural que esteja disponível como um projeto de código aberto. O próprio jogo resultado do **Projeto Coma**, desde seu início, teve seu código hospedado no **Google Code**[1], destinado a projetos de código aberto.

Desta forma, a licença da biblioteca MAC~CO, é a ***General Public License version 3 (GPL 3)***, da ***Free Software Foundation***, disponível em: http://www.gnu.org/licenses/gpl.html.

1 http://abre.la/Coma_SVNtrunk

Preparando-se

Pré-requisitos

Antes de utilizar a biblioteca MAC-CO, certifique-se de atender ao requisito de uso da *engine* de desenvolvimento de jogos Unity3D, na versão 4.3.1, pelo menos, que pode ser obtida em: http://unity3d.com.

Implantando a biblioteca

Obtendo

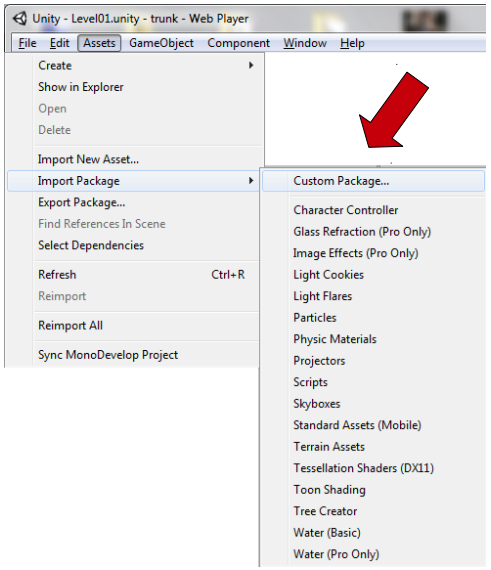
A biblioteca pode ser baixada em http://mac-co.jonasluz.com.br.

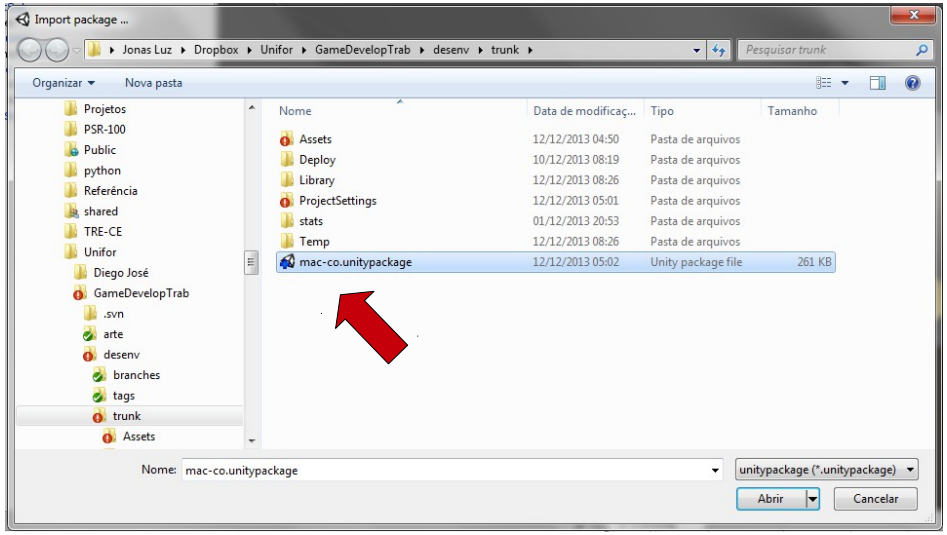
Estretanto, se estiver trabalhando no Projeto Coma, a biblioteca MAC-CO já está inclusa no código e nada além precisa ser baixado.

Instalando

A biblioteca é distribuída como um pacote Unity, que deve ser importado para seu projeto Unity. Isto pode ser feito através do menu **Assets → Import Package → Custom Package**. Por meio da caixa de abertura de arquivos, selecione o `mac-co.unitypackage`.

Novamente, isto não é necessário se estiver trabalhando com o código do Projeto Coma, no qual a biblioteca já está inclusa.





Conhecendo a biblioteca

MAC~CO possui, na atual versão, um acoplamento bastante forte com a estética e jogabilidade do Projeto Coma, jogo para o qual a biblioteca está sendo desenvolvida.

Assim, o uso de MAC~CO não é indicado em jogos cuja estética ou jogabilidade sejam muito diferentes do Projeto Coma, sua referência.

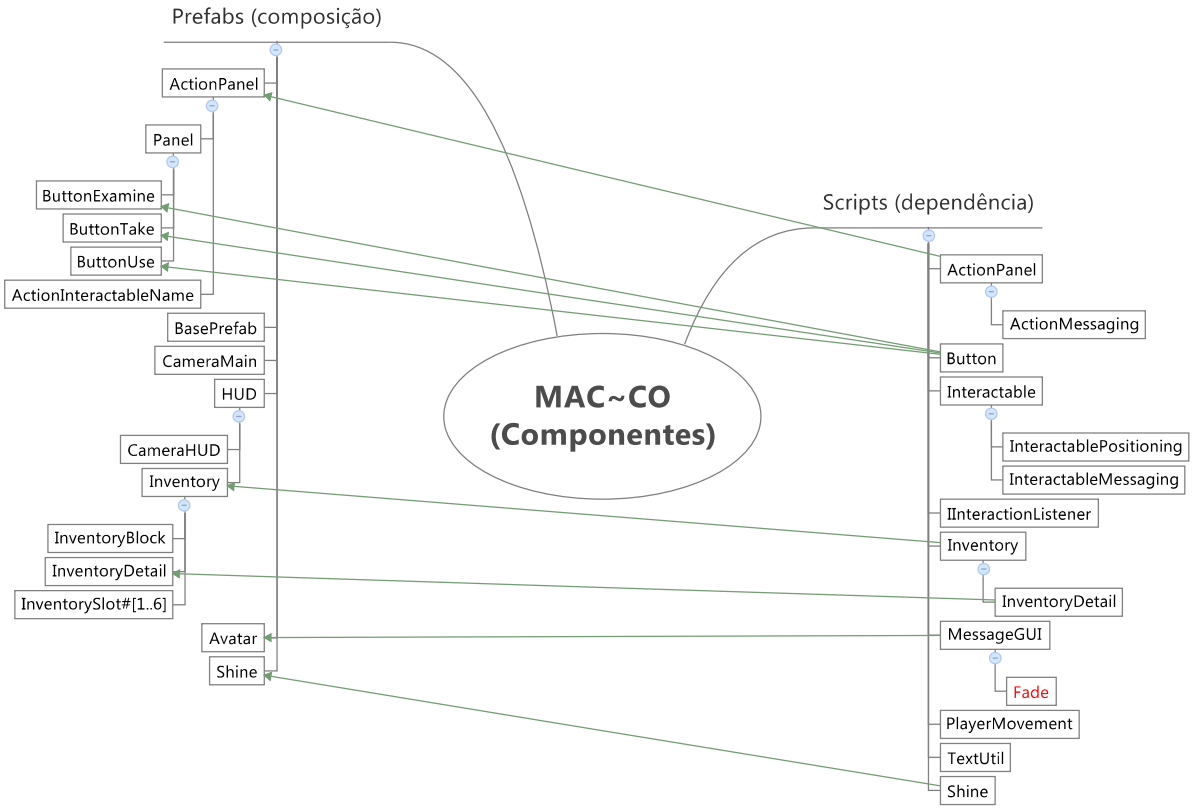


Ilustração 1: Mapa mental - Arquitetura de componentes da MAC~CO.

Alerta feito, o uso da biblioteca requer o conhecimento prévio de seus elementos componentes, o que será explicado nesta seção.

São dois grandes grupos que compõem toda a biblioteca: os prefabs e os scripts.

Os Prefabs

Os prefabs são os objetos visuais predefinidos da biblioteca, e que devem ser utilizados no jogo.

É basicamente nos prefabs que reside o forte acoplamento, já citado, com a estética do Projeto Coma. Em uma versão futura da biblioteca, esta dependência deverá ser eliminada, o que deverá ser feito através da adoção de uma abordagem diferente em relação aos prefabs; uma opção seria substitui-los por elementos vazios – "*empty*" na nomenclatura da Unity Engine –, mas a melhor abordagem, com certeza, seria compor a biblioteca MAC~CO apenas de scripts adaptáveis a quaisquer objetos gráficos que o desenvolvedor resolva usar.

Atualmente, os prefabs são:

- `ActionPanel`: o painel de ações, que aparece quando o jogador seleciona um item com o qual pode interagir, dando-lhe três opções: "examinar" (botão verde), "pegar" (botão azul) e "usar" (botão vermelho). O Action Panel é, na verdade, um *empty* da Unity, que agrupa:
  - `Panel`: o painel em si com os três botões descritos, aos quais é associado o script `Button.js`; e
  - `ActionInteractableName`: texto 3D que exibe o nome do objeto "interagível" selecionado pelo jogador.

Ilustração 2: O prefab ActionPanel.

O `ActionPanel` está associado aos scripts `ActionPanel.js` e `ActionMessaging.js`, este último dependência do primeiro.

- `Avatar`: **GUITexture** que deve ser associada à textura da imagem do avatar do personagem. Está associado a dois scripts: `MessageGUI.js` e **`Fade.js`**, sendo este último dependência do primeiro e um script de terceiro, creditado no código fonte, utilizado para o efeito de fade-in e fade-out da caixa de texto.
- `CameraMain`: Câmera pré-configurada, aplicada em substituição à câmara principal na cena. Recebe o AudioSource da trilha sonora e o script implementador da interface `IinteractionListener`.

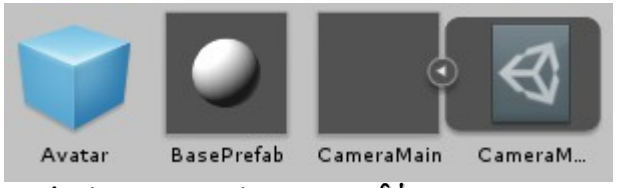


Ilustração 3: Avatar, BasePrefab e CameraMain

- `BasePrefab`: Marcador de base de movimentação do personagem, deve ser colocado no centro do cenário, de forma que o personagem possa, se necessário referenciá-lo durante sua movimentação e, a partir dele, alcançar qualquer lugar do cenário. É uma solução "suja" para resolver o problema de *pathfinding*, o que é, com certeza, algo a melhorar em versão futura da mecânica.
- `HUD`: Heads-up display, um *empty* da Unity que agrupa:
  - `CameraHUD`: câmera com imagem sobreposta à da `CameraMain`, mas especializada em exibir o HUD do jogo.
  - `Inventory`: o inventário do jogo, onde o personagem guarda os objetos "pegos" (botão azul) na cena. Possui associado o script `Inventory.js` e é composto por:
    - `Inventory_Block`: bloco do inventário que cobre a cena, evitando interações indesejadas com esta quando do detalhamento de objetos no inventário.
    - `Inventory_Detail`: marca a posição e rotação – por meio do script `InventoryDetail.js` que lhe é associado – do objeto que é detalhado no inventário.

Ilustração 4: Os prefabs HUD e Shine.

- `Shine`: O brilho que surge nos objetos passíveis de interação pelo jogador. Este brilho é acionado sobre o objeto "interagível" quando o jogador passa o ponteiro do *mouse* sobre este.

Os Scripts

É nos scripts, escritos no dialeto JavaScript da Unity, que está a lógica funcional da mecânica de apontar e clicar. Em versão futura da biblioteca MAC~CO, esta deve ser composta de um conjunto de scripts, reduzindo-se ao mínimo possível sua dependência de prefabs. Isto dará ao programador a liberdade de utilizar a biblioteca em quaisquer objetos e com a estética que lhe convier, inclusive 2D.

Os scripts que compõem a MAC~CO atualmente são:

- `ActionPanel.js`: Implementa as funcionalidades do painel de ações ("examinar", "pegar" e "usar") que ocorrem em cena. Já vem aplicado ao prefab `ActionPanel` e não precisa ser mexido pelo programador da cena. Implementa o padrão de projeto **singleton**, o que significa que somente pode existir uma instância sua em cada cena (o que é óbvio: há apenas um painel de ações por cena). Requer o `ActionMessaging.js`.
- `ActionMessaging.js`: Mantém a lista de mensagens que devem ser exibidas quando houver tentativa de interação frustrada pelo jogador (tentar "pegar" um armário pesado, por exemplo). Estas mensagens são mantidas em um arquivo-texto, o que facilita sua edição por outro membro da equipe de desenvolvimento que não seja o programador. Já vem aplicado ao prefab `ActionPanel` e não precisa ser mexido pelo programador da cena.
- `Button.js`: Controla a identificação das ações de cada botão ao qual é aplicado. Os três botões do prefab `ActionPanel` já possuem o scripts devidamente associado e configurado a cada ação específica ("examinar", "pegar" e "usar"), o que dispensa a intervenção do programador da cena.
- `MessageGUI.js`: Mantém a caixa de mensagens que é exibida na interação do jogador com os objetos na cena. Já está previamente associado ao prefab Avatar, não havendo necessidade de ser mexido pelo programador da cena. Requer `Fade.js`.
- `Fade.js`: Código de terceiro, desenvolvido por Eric Haines e disponível em http://www.dmu.com/unity/ref6.html. Utilizado para os efeitos de fade in e fade out aplicados à caixa de mensagens.
- `TextUtil.js`: Classe estática que mantém o método `ReadLines(res:TextAsset):String[]`, que faz a leitura dos arquivos-texto de configuração e retorna sua lista de linhas. É utilizado por diversos outros scripts para a carga de listas de mensagens. Não precisa ser mexido pelo programador da cena.

- `Interactable.js`: Script que deve se aplicado pelo programador da cena a cada objeto passível de interação pelo jogador ("interagível"). A técnica de aplicação do script será vista mais adiante neste manual. Requer outros dois scripts, aplicados automaticamente pelo Unity quando `Interactable.js` é aplicado: `InteractableMessaging.js` e `InteractablePositioning.js`.

- `InteractableMessaging.js`: Controla as mensagens de um objeto "interagível" específico. É aplicado automaticamente pela Unity quando da aplicação de `Interactable.js`. As mensagens de cada objeto, bem como seu nome em cena e descrição longa – vista pelo jogador no detalhamento do objeto no inventário –, se for o caso, são mantidas por este script, que as lê de um arquivo-texto específico por objeto "interagível". Esta abordagem facilita a edição das mensagens por um membro da equipe de desenvolvimento que não seja o programador de cena.

- `InteractablePositioning.js`: Aplicado automaticamente pela Unity quando da aplicação de `Interactable.js`, este script controla o posicionamento, escala e rotação de cada objeto "interagível", ajustando-o:
  - em cena, na posição, escala e rotação originalmente postas pelo programador de cena;
  - em uma célula do inventário se o objeto puder ser e for "pego" pelo jogador;
  - em destaque no modo de detalhamento do inventário; ou
  - no lugar do ponteiro do mouse se o objeto estiver sendo "usado" pelo jogador.

- `Inventory.js`: Implementa a funcionalidade do inventário, vem previamente aplicado ao prefab `Inventory`, dispensando a intervenção do programador de cena. Também implementa o padrão de projeto **singleton** (há apenas um inventário por cena).

- `InventoryDetail.js`: Implementa a rotação do objeto interagível trazido para o modo de detalhamento dentro do inventário. Já está previamente aplicado ao prefab `Inventory_Detail` e não precisa ser alterado pelo programador de cena.

- `IInteractionListener.js` (interface): Define a interface IInteractionListener, que deve ser implementada em uma classe pelo programador da cena, com os métodos:
  - `NotifyInteraction(interactable:Interactable)`: que é chamado pela MAC~CO quando houver interação com algum objeto "interagível" da cena.
  - `NotifyUse(used:Interactable, over:Interactable)`: que é chamado pela MAC~CO quando há interação de uso (verbo "usar") de um objeto (`used`) sobre outro (`over`).

  Sua implementação será explicada em detalhes mais adiante.

- `Shine.js`: Implementação da funcionalidade do brilho de destaque dos objetos "interagíveis", já vem aplicado ao prefab Shine e não requer intervenção do programador da cena.

- `PlayerMovement.js`: Controla a movimentação do personagem em cena e implementa – de forma "suja" ainda, dependendo da existência em cena da BasePrefab – o *pathfinding*. Deve ser aplicado pelo programador de cena ao personagem do jogo, o qual não pode conter outro componente de física da Unity que não seja o **CharacterController**. Este, aliás, é requerido por `PlayerMovement.js`, que o aplica automaticamente se não existir.

A ilustração 5 mostra o diagrama de classes (UML) da implementação da biblioteca MAC~CO. Adicionalmente, constam no diagrama as classes implementadas para a cena da Fase 1 do Projeto Coma, o que serve para ilustrar como as classes a serem implementadas pelo programador de cena devem se integrar à biblioteca.

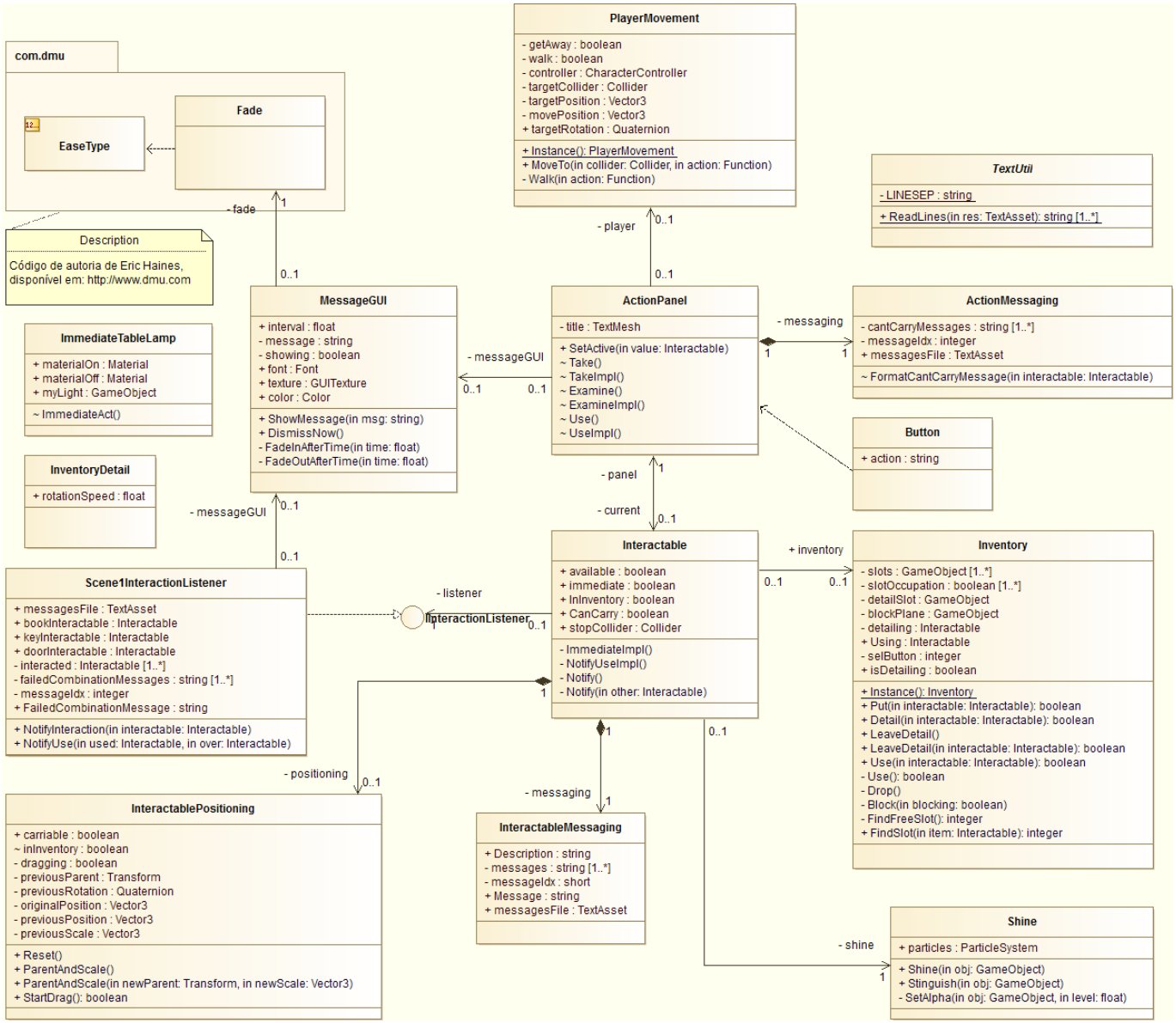


Ilustração 5: Diagrama de classes da biblioteca MAC~CO, incluindo a cena da Fase 1 do Projeto Coma.

Usando a biblioteca

Esta seção explicará o uso da biblioteca passo a passo. Aqui, parte-se da premissa de que a biblioteca já foi importada no seu projeto ou de que se está trabalhando com o Projeto Coma. Caso não o tenha feito ainda, consulte a seção Implantando a biblioteca , à página 5.

Preparando a Cena

1. Crie uma nova cena.
2. Exclua da cena a câmera principal (Main Camera).
3. Adicione à cena os seguintes prefabs da biblioteca: `ActionPanel`, `Avatar`, `CameraMain`, `HUD` e `Shine`.

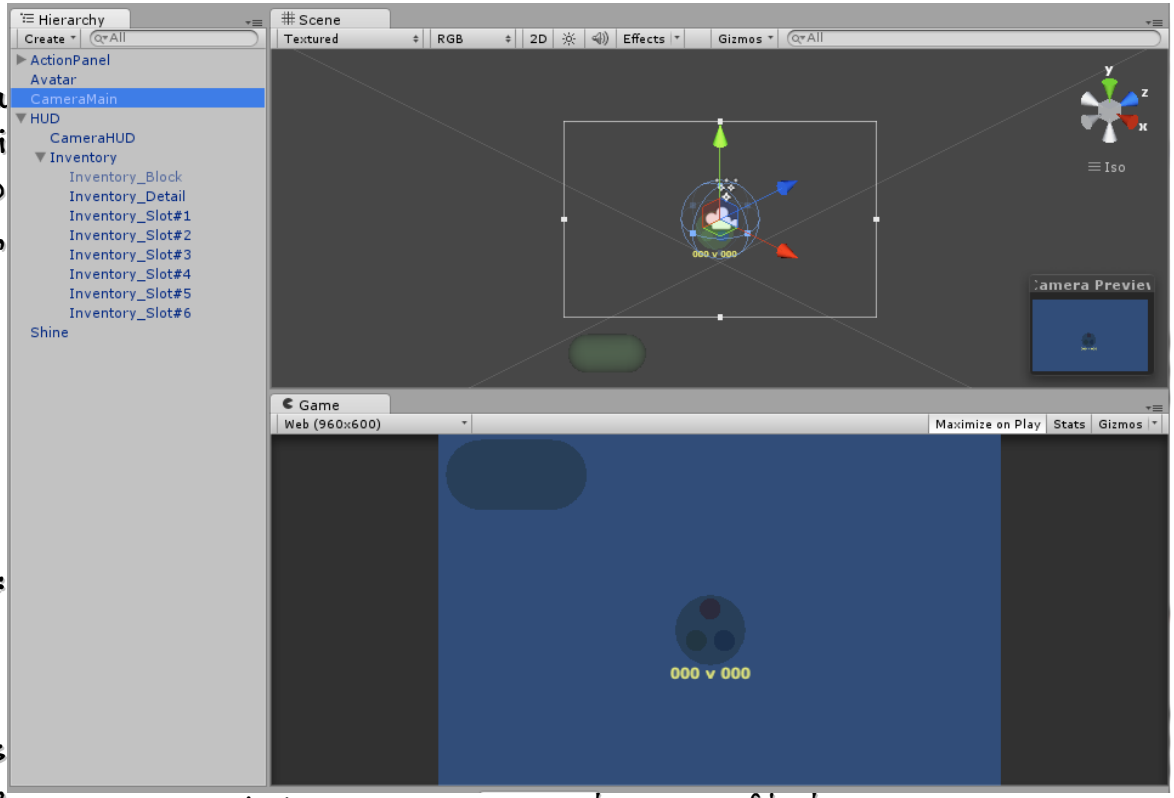


Ilustração 6: Uma cena preparada com os prefabs da MAC~CO.

Observe que a cena passa a contar com duas câmeras: `CameraMain`, que mostra a cena em si e que deve ser usada como referência para posicionar o cenário, e a `CameraHUD`, subordinada ao prefab `HUD`. A imagem desta última sobrepõe a da primeira sem apagá-la. Isto permite que se possa mexer no cenário sem ter que se preocupar com os elementos 3D do HUD. Entretanto, por esta razão, o plano de bloqueio do inventário (`Inventory_Block`), impedem que se tenha uma boa visão da `CameraMain` na janela Game. Para se evitar isto, pode-se, em tempo de desenvolvimento, desativar o objeto de cena `Inventory_Block`.

Adicionando o cenário

1. Importe seu cenário para o projeto, se ainda não o tiver feito.
2. Arraste seu cenário para a cena, posicionando adequadamente de maneira que possa ser visto pela `CameraMain`. Sugere-se que o centro do cenário fique posicionado em (0, 0, 0).

Adicionando o personagem

1. Importe o personagem para o projeto, se ainda não o tiver feito.
2. Arraste seu personagem para a cena, posicionando-o adequadamente no cenário.
3. Retire do personagem os componentes de física existentes.
4. Adicione ao personagem o componente script `PlayerMovement.js`, da MAC~CO.
5. Certifique-se de que, automaticamente, tenha sido adicionado também o componente de física da Unity `CharacterController`, uma dependência de `PlayerMovement`. Senão, adicione-o manualmente.
6. Ajuste as dimensões do componente `CharacterController` de acordo com as de seu personagem.
7. Marque o personagem com a tag Player, criando-a se não existir.

A Ilustração 7 mostra uma cena preparada com os prefabs do MAC~CO. Deve-se observar a possibilidade de ajustes de escala do personagem e as dimensões do componente `CharacterController`.

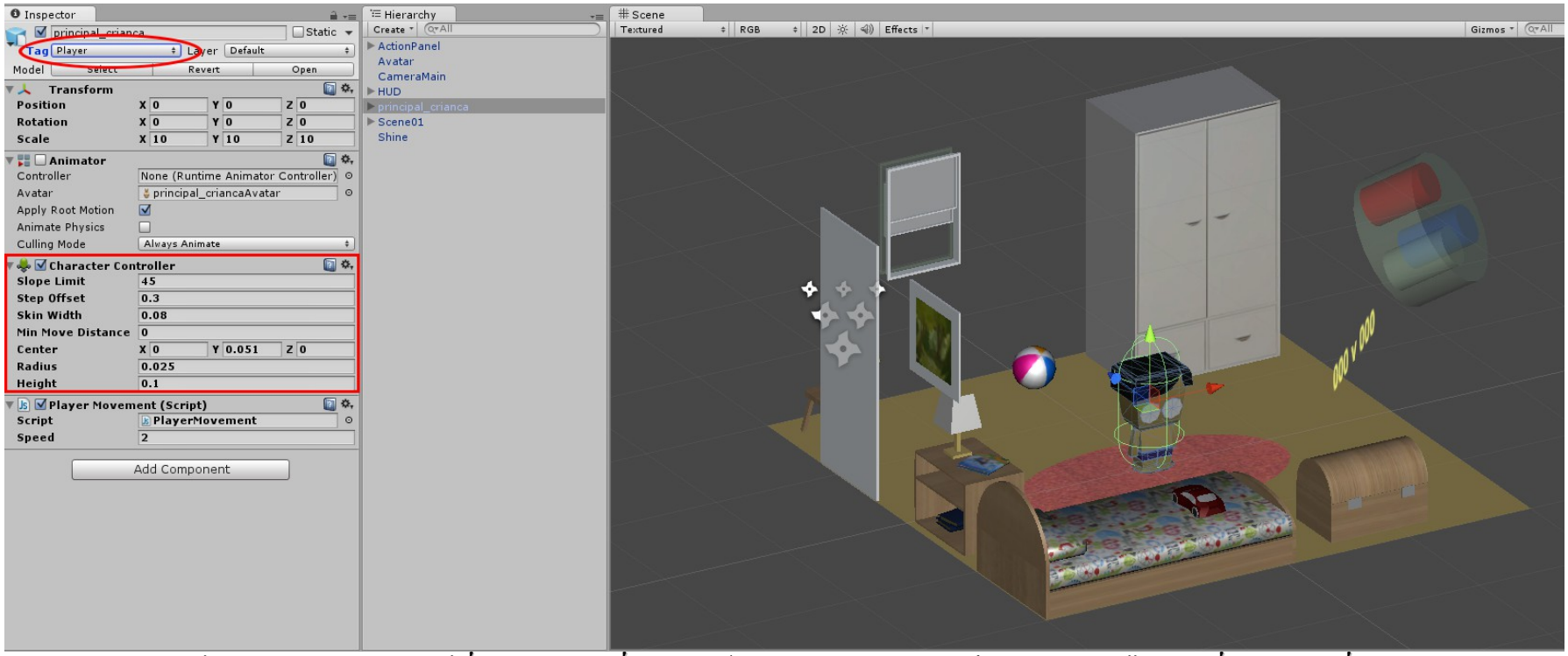


Ilustração 7: Visão parcial de cena preparada com cenário e personagem, cujo CharacterController teve de ser ajustado.

A interação com os objetos

A interação no cenário

A interação do personagem com os objetos da cena acontece por meio do clique do mouse, que aciona o painel de ações. Nem todos os objetos da cena permitem interação, sendo responsabilidade do programador configurar cada um dos objetos "interagíveis". Estes se classificam em dois tipos:

- Os de ação imediata, que executam apenas uma ação, acionada imediatamente após o clique sobre o objeto; exemplo: um abajur que, ao ser clicado, acende ou apaga.
- Os demais interagíveis, que permitem a escolha da ação, através do painel de ações, que pode ser:
    - EXAMINAR: o personagem principal vai até o objeto e o examina, relatando ao jogador o que vê. A cada acionamento desta ação, uma frase descritiva é falada pelo personagem; quando as frases sobre aquele objeto se esgotam, estas se repetem a partir da primeira.
    - PEGAR: o personagem principal vai até o objeto e o "pega" para carregar no inventário. Ao ser acionada esta ação, MAC~CO retira o objeto do cenário e o coloca em um dos espaços livres no inventário; caso não haja, o personagem notifica o jogador que não pode carregar mais nada.
    - USAR: o personagem principal vai até o objeto e o "pega" para uso sobre outro. Ao ser acionada esta ação, o objeto substitui o ponteiro do mouse, devendo ser "apontado" para outro objeto e o clique acionado. Isto informa à MAC~CO que se pretende usar um objeto com outro. A engine responde notificando a interação à classe que implementa a interface `IINteractionListener`, responsável pela execução do "roteiro" da cena. Mais detalhes sobre isto adiante.

A distinção entre os objetos de ação imediata e os demais se dá através da atribuição de valor à propriedade booleana (aceita valores verdadeiro ou falso) `Immediate`, disponível no componente `Interactable`. Além desta propriedade, os objetos "interagíveis" possuem ainda outras duas propriedades booleanas úteis:

- `Available`: indica que o objeto "interagível" está disponível. Pode ser marcada com valor "falso" para se impedir a interação do jogador com um determinado objeto até que este possa ser "percebido" pelo personagem principal. Esta propriedade também pode ser encontrada no componente `Interactable`.
- `Carriable`: indica que o objeto pode ser "carregado" no inventário. Há objetos que devem ser interagíveis, mas não deveriam poder ser carregados; exemplos típicos são: uma porta, uma janela ou um guarda-roupa. Esta propriedade está disponível no componente `InteractablePositioning`, dependência de `Interactable`.

A interação no inventário

Os objetos "interagíveis" que podem ser carregados (propriedade `Carriable` com valor "verdadeiro"), ao terem ativada a ação PEGAR no painel de ações e tendo espaço disponível, são colocados no inventário. O jogador pode, então, com o mouse, ativar estes objetos que o personagem está carregando. Ao ativar um dos objetos no inventário, este é "aberto" sobrepondo o cenário. Clicar sobre o objeto em destaque fecha o inventário, trazendo de volta o cenário.

A ilustração 8 mostra o inventário aberto com um objeto – um livro – em destaque. O ponteiro do mouse está sobre outro objeto – a bola – que, se acionado, assumirá a posição de destaque no inventário.

A interação com qualquer objeto em destaque no inventário se dá através dos botões superiores. Além do primeiro, que apenas mostra o nome do objeto em destaque, são três botões que correspondem a três ações:

- EXAMINAR: semelhante à opção de examinar disponível no painel de ações no cenário, esta ação dá ao jogador uma descrição detalhada do objeto em destaque, descrição esta que não está disponível no cenário – afinal, com o objeto no inventário, o personagem pode examinar o objeto com mais atenção!
- SOLTAR: ação oposta à ação PEGAR disponível no painel de ações do cenário, esta ação retorna o objeto em destaque à sua posição original no cenário e fecha o inventário.
- USAR: possui o mesmo efeito da ação homônima disponível no painel de ações do cenário. Ao ser acionada, fecha o inventário e substitui o ponteiro do mouse pelo objeto anteriormente em destaque, permitindo que este seja utilizado pelo personagem com outro objeto do cenário ou do inventário.

Ilustração 9: O painel de ações aberto sobre um objeto "interagível" no cenário.

Uma vez compreendidas as formas de interação possíveis do personagem com os objetos, resta saber como estas interações e as mensagens dos objetos são programadas com a MAC~CO.

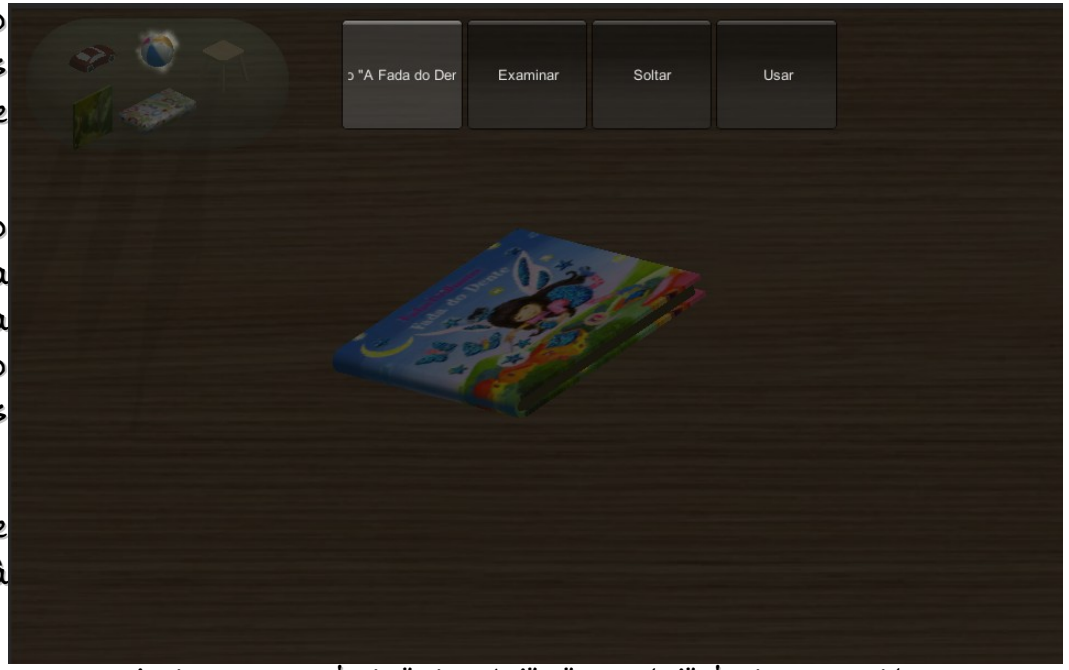


Ilustração 8: Um objeto "interagível" e "carregável" aberto no inventário.

Os objetos "interagíveis"

O arquivo de definição

Para cada objeto "interagível" deve ser editado um arquivo-texto com sua definição, que inclui: nome, as mensagens a serem exibidas no acionamento da ação de EXAMINAR no cenário e a descrição destalhada, exibida no acionamento da ação de EXAMINAR no inventário.

O arquivo deve seguir o formato indicado a seguir:

1. Primeira linha: nome do objeto.
2. Segunda linha: número inteiro *n*, indicando a quantidade de linhas seguintes que são mensagens a exibir no EXAMINAR do cenário.
3. Linhas 3 a n+2: cada linha, uma mensagem a ser exibida no EXAMINAR do cenário. As mensagens são exibidas na sequência, e repetidas quando se esgotam.
4. Linhas n+3 adiante: descrição detalhada do objeto, que pode estar em múltiplas linhas. É exibida pela MAC~CO quando é acionada a ação EXAMINAR no inventário. Caso o objeto não seja "carregável", deve ser mantida, pelo menos, uma linha em branco.

A definição dos objetos "interagíveis" em arquivos-texto separados do código, além de dispensar o uso do Unity para esta tarefa, permite que tal definição seja feita por um membro da equipe não programados, como o responsável pelo roteiro do jogo, por exemplo.

Abaixo, um exemplo de arquivo de definição para um objeto "interagível" "carregável"; uma **chave**:

```
chave
3
Finalmente, achei a chave!
Tão brilhante e dourada, minha chave!
Acho que ficar olhando para ela não vai me ajudar a usá-la!
Esta é uma bela chave dourada.
E é exatamente do que preciso agora.
Só me resta utilizá-la no lugar certo!
```

Agora, um exemplo diferente: um arquivo de definição de um objeto "interagível" <u>não</u> "carregável" – observe-se que não há descrição detalhada para o referido objeto, um guarda-roupa:

```
guarda-roupa
3
Meu guarda-roupa deve estar uma bagunça.
Procurar alguma coisa aí dentro seria como tentar encontrar
uma agulha num palheiro.
Acho que estou sem meias limpas.
-
```

A movimentação do personagem até os objetos

Ao se acionar uma ação sobre um objeto "interagível" no cenário, o personagem precisa ir até o objeto. Considerando a simplicidade dos cenários originalmente propostos no Projeto Coma – veja detalhes na seção "O Projeto Coma" à página 4 – e o curto prazo que se tinha para o desenvolvimento do jogo, deu-se à movimentação do personagem uma solução simples, que envolve a definição, para cada objeto "interagível" de um `Collider` (da Unity) cuja posição o personagem tenta alcançar, mas pára ao colidir com este `Collider`, cuja indicação se dá através da propriedade `StopCollider`, do componente `Interactable`.

A escolha do `Collider` parte da escolha do seu `GameObject`, devendo-se selecionar um cuja posição o personagem possa atingir sem ficar preso no caminho a partir de qualquer outra posição. Este objeto pode até mesmo ser o próprio objeto "interagível", mas a melhor forma de se fazer esta escolha é através de tentativa e erro. Se não houver um objeto adequado na cena, a alternativa é se criar um objeto "invisível" – ou seja, cujo `MeshRenderer` seja desativado – com `Collider` e indicar este último. A ilustração 10 mostra este caso.

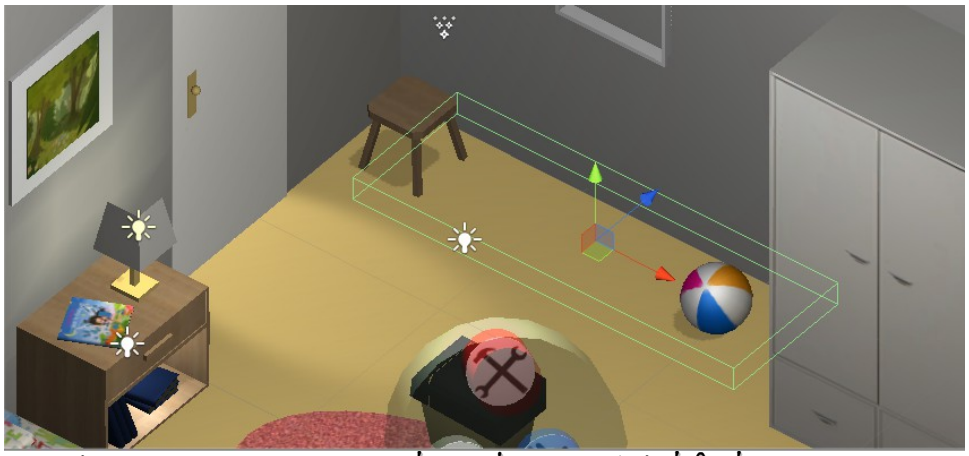


Ilustração 10: `Collider` criado em objeto invisível, definido como `Stop Collider` de três "interagíveis" distintos: o banco, a bola e a janela.

Configurando os "interagíveis"

Eis o passo a passo para se configurar um "interagível" na cena:

1. Selecione o objeto que deve permitir a interação com o jogador. Marque-o com a tag `Interactable`, criando-a se não existir.
2. Atribua ao objeto o componente `Interactable` da MAC-CO.
3. Verifique se foram incluídos ao objeto, automaticamente, os componentes adicionais dos quais `Interactable` depende: `InteractableMessaging` e `InteractablePositioning`. Se não, adicione manualmente os referidos componentes.
4. Defina, na propriedade `StopCollider` do componente `Interactable`, qual o `Collider` cuja posição deve ser considerada pelo personagem ao tentar alcançar o objeto.
5. Indique o arquivo de definição do objeto – sobre isto, leia a seção "O arquivo de definição" à página 15 – no atributo `MessageFile` do componente `InteractableMessaging`.
6. Decida se este objeto específico estará disponível inicialmente e defina o valor da propriedade `Available` do componente `Interactable` conforme o caso. Isto pode ser mudado em tempo de execução por meio da programação da lógica do jogo.

7. Decida se o objeto é de ação imediata – para mais detalhes , leia a seção "A interação no cenário" à página 13 – e, se for o caso, marque como verdadeiro o atributo **`Immediate`**. Vá ao passo 9.
8. Decida se o objeto pode ser carregado pelo personagem no inventário. Atribua o valor correspondente à propriedade **`Carriable`** do componente **`InteractablePositioning`**.
9. Repita todos os passos para cada objeto "interagível" na cena.

No caso dos objetos de ação imediata (**`Immediate`** igual a verdadeiro), quando estes forem acionados (clicados com o *mouse*) pelo jogador, MAC-CO fará uma chamada ao método **`ImmediateAct()`** onde quer que ele esteja no contexto do objeto "interagível" selecionado. Desta forma, o objeto deve contar com um componente script que implemente tal método. Este script deve ser implementado pelo programador da cena e seu comportamento depende, obviamente, da ação a ser executada pelo objeto "interagível" de ação imediata.

Exemplifica-se, abaixo, um script que poderia ser usado para um abajur (objeto "interagível" de ação imediata) que, ao ser acionado, acende se estiver apagado e apaga se estiver aceso.

```
/**
 * ImmediateTableLamp.js
 * Immediate functionality for table lamp (level 01).
 * By Jonas Luz, Oct-Nov 2013
 **/

#pragma strict

public var materialOn : Material;
public var materialOff : Material;
public var myLight : GameObject;

function Start () {
      myLight.SetActive(false);
}

// Imediate interaction with table lamp.
public function ImmediateAct() {
      myLight.SetActive( ! myLight.activeSelf );
      renderer.material = myLight.activeSelf ? materialOn :
materialOff;
}
```

Neste ponto, a programação da cena está quase completa. Esta já conta com o cenário, o personagem e os objetos com os quais este pode interagir no cenário ou no inventário. O passo final é se programar o fluxo de andamento do roteiro do jogo, ou seja, a lógica de resposta do jogo às ações do jogador de acordo com o que estiver definido no roteiro. Para isto, deve-se implementar a interface **`IInteractionListener`**, o que é objeto de estudo da próxima seção.

Programando o roteiro da cena

A interação do personagem com os objetos de cena, sejam estes de ação imediata ou não, passíveis de serem carregados no inventário ou não; esta interação se torna apenas uma brincadeira se não houver um objetivo. É a busca do alcance deste objetivo pelo jogador que dá à cena sua natureza de "jogo": a cena precisa ser resolvida.

A biblioteca MAC~CO facilita o trabalho do programador também aqui, notificando uma determinada classe de componente, implementada pelo programador, de todas as interações havidas na cena e no inventário pelo jogador. Para tanto, esta classe de componente deve materializar a interface **`IInteractionListener`**, cuja definição é colocada abaixo:

```
/**
 * InteractionListener.js
 * Object interaction listener interface.
 * By Jonas Luz, Nov 2013.
 **/

#pragma strict

public interface IInteractionListener {
      function NotifyInteraction(interactable:Interactable);
      function NotifyUse(used:Interactable, over:Interactable);
}
```

Como se pode ver, são dois os métodos ativados pela MAC~CO quando ocorre uma interação na cena:

- **`NotifyInteraction(interactable)`**: chamado pela MAC~CO quando o jogador interage com qualquer objeto "Interagível" passando para este o componente **`Interactable`** do objeto.
- **`NotifyUse(used, over)`**: chamado pela MAC~CO quando o jogador "usa" um objeto sobre outro. São passados os dois componentes **`Interactable`** dos objetos envolvidos, sendo:
  - **`used`**: o **`Interactable`** do objeto que está sendo "usado"; e
  - **`over`**: o **`Interactable`** do objeto sobre o qual **`used`** foi "usado".

Com estas notificações cabe ao programador implementar toda a lógica necessária para manter o estado de progresso da cena.

A classe que implementa **`IInteractionListener`** deve ser colocada como componente em algum objeto do jogo (**`GameObject`**), que pode ser um objeto vazio (empty) criado especialmente para isto ou a câmera principal. Então, em seu método **`Awake()`**, deve se inscrever como a classe auscultadora das interações, chamando o método de classe **`Interactable.SetListener(IInteractionListener)`**:

```
function Awake() {
      Interactable.SetListener(this as IInteractionListener);
}
```

O código listado adiante exemplifica uma classe que implementa **IInteractionListener**. O exemplo mostrado traz a codificação da lógica da primeira fase do jogo do Projeto Coma – para detalhes, leia a seção "O Projeto Coma", à página 4 – e portanto, traz a solução da fase; daí, sugere-se cautela para quem não deseja receber *spoilers*. ;)

```
/**
 * Scene1InteractionListener.js
 * Interaction Listener for Scene1.
 * By Jonas Luz, Nov. 2013.
 **/

#pragma strict

public class Scene1InteractionListener extends MonoBehaviour
implements IInteractionListener {

      // messages of failed combinations.
      public var messagesFile:TextAsset;

      // message GUI.
      public var messageGUI:MessageGUI;

      // interactable of the book,
      // to be activated only after two interactions.
      public var bookInteractable:Interactable;

      // interactables of door and key,
      // that must be combined to win the level.
      public var doorInteractable:Interactable;
      public var keyInteractable:Interactable;

      // interaction counter.
      private var interacted:Array = Array();

      // failed combination messages.
      private var failedCombinationMessages:String[];
      private var messageIdx:short;

      private function get FailedCombinationMessage() {
            if (failedCombinationMessages.Length == 0)
                  return null;
            if (messageIdx == failedCombinationMessages.Length)
                  messageIdx = 0;
            return failedCombinationMessages[messageIdx++];
      }

      function Awake() {
            // subscribe to notifier.
            Interactable.SetListener(
                  this as IInteractionListener);
            // Read failed combination messages.
            FailedCombinationMessages =
            TextUtil.ReadLines(messagesFile);
```

```
    }

    function Start() {
        bookInteractable.available = false;
    }

    function NotifyInteraction(interactable:Interactable) {
        if (interactable in interacted) return;
        interacted.Add(interactable);
        if (interacted.length == 2)
            bookInteractable.available = true;
    }

    function NotifyUse(used:Interactable, over:Interactable) {
        if (used == keyInteractable
            && over == doorInteractable) {
            Application.LoadLevel("Level01_");
        } else {
            var msg:String =
             String.Format(
                  FailedCombinationMessage,
                  used.gameObject.name,
                  over.gameObject.name);
            messageGUI.ShowMessage(msg);
        }
    }
}
```

No código exemplificado, são mantidas as seguintes variáveis públicas, definidas na programação da cena:

- **`messagesFile:TextAsset`** – define o arquivo onde estão configuradas as mensagens de falha de interação, utilizadas para notificar o jogador, por exemplo, de que tentar usar a chave com o baú não traz resultados.
- **`MessageGUI:MessageGUI`** – componente da MAC~CO que exibe as mensagens ao jogador, ou seja, os pensamentos do personagem.
- **`BookInteractable:Interactable`** – componente **`Interactable`** do livro de estórias existente na cena. A interação do jogador com este objeto livro está disponível (atributo **`Available`** falso) até que este tenha interagido com, pelo menos, dois outros objetos.
- `doorInteractable:Interactable` e `keyInteractable:Interactable` – os componentes Interactable, respectivamente, da porta e da chave, ambos elementos da cena. A interação de uso de um objeto sobre o outro deverá disparar o final da cena.
- **`interacted:Array`** – array de Interactable que guarda os objetos com os quais o jogador já interagiu. Isto é necessário para se liberar, no momento certo, a possibilidade de interação com o livro presente na cena (veja variável **`BookInteractable`**, acima).
- `failedCombinationMessages:String[]` e `messageIdx:short` – array de mensagens de aviso de falha de combinação e índice de navegação neste array, respectivamente.

Também é mantida uma propriedade de classe:

- `FailedCombinationMessage` – retorna a próxima mensagem de alerta de falha de interação de uso, que é escolhida sequencialmente dentre aquelas disponíveis no array `failedCombinationMessages` (vide descrição acima).

A implementação do método **`Awake()`**, especificado na superclasse **`Behaviour`**, da Unity, faz o seguinte:

1. registra a presente instância da classe como a auscultadora das interações usando **`Interactable.SetListener`**.
2. Faz a leitura das mensagens de erro de interação, a partir do arquivo indicado em **`messagesFile`**, e as guarda no array `failedCombinationMessages`.

A implementação do método **`Start()`**, especificado na superclasse **`Behaviour`**, da Unity, inicializa o **`Interactable`** do livro presente na cena, configurado na variável **`bookInteractable`**, deixando-o indisponível para interação do jogador.

Eis o que faz a implementação do método **`NotifyInteraction`**, exigida pela nossa interface **`IInteractionListener`**:

1. Caso o objeto com o qual o jogador interagiu não tenha ainda sido contado, o é através da inclusão deste no array **`interacted`**.
2. Caso o número de objetos distintos com os quais o jogador interagiu tenha atingido a quantidade de dois, o livro presente na cena, cujo **`Interactable`** foi indicado na variável **`bookInteractable`**, fica disponível para interação com o jogador.

A interface **`IInteractionListener`** determina também a implementação do método **`NotifyUse`**, que, o exemplo mostrado, faz o seguinte:

1. Verifica se a chave está sendo usada sobre a porta. Em caso afirmativo, o jogador passou de fase, e é carregada a cena "**`Level01_`**".
2. Se a interação de uso não corresponder à chave sendo usada sobre a porta, então é exibida uma mensagem de interação sem resultados dentre as disponíveis, através da chamada da propriedade `FailedCombinationMessage`.

Conclusão

A biblioteca MAC~CO, em sua versão atual, implementa uma mecânica de "apontar e clicar" ("*point'n'click*") simples, mas adequada à implementação da primeira fase do jogo do Projeto Coma (detalhes na seção "O Projeto Coma", à página 4), o que ficou definido como prioridade de implementação desde o início do projeto.

As principais funcionalidades atualmente disponíveis são:

- A movimentação do personagem no cenário até os objetos.
- A interação do personagem com os objetos no cenário e no inventário, com a flexibilidade de haverem objetos de ação imediata, com disponibilidade variável, e objetos fixos no cenário.
- A implementação de ações sobre os objetos: EXAMINAR, PEGAR/SOLTAR e USAR, no cenário ou no inventário.
- Notificação das interações a uma classe, definida pelo programador, que se encarregará de processar a lógica do estado e evolução da cena/fase do jogo.

A biblioteca pode e deve ser evoluída. Dentre as funcionalidades não presentes que se imagina serem úteis, até mesmo em outras fases previstas para o jogo do Projeto Coma, está a que envolve o diálogo do personagem principal com outro personagem.

Sugestões e críticas construtivas são sempre bem-vindas.

Ilustração 11: Uma cena da primeira fase do jogo do Projeto Coma.

Universidade de Fortaleza – Unifor
Especialização em Animação e Jogos Eletrônicos
Turma 2012-2013
Disciplina: Game Development
Prof. Daniel Siqueira

Implementação de uma mecânica "aponte e clique" para o jogo do Projeto Coma.

Jonas de A. Luz Jr. <contato@jonasluz.com>

http://mac-co.jonasluz.com.br
http://abre.la/ProjetoComa

Fortaleza, Janeiro de 2014