DCC192

2025/2

UFMG

Desenvolvimento de Jogos Digitais

A24: Cutscenes e Narrativas

Prof. Lucas N. Ferreira

Plano de Aula

- Cutsenes
  - Pré-renderizadas
  - Renderizadas em Tempo Real
    - Hardcoded
    - Interpretadores
    - Editores Gráficos
- Sistemas de Missões
  - Representação
  - Finalizando Eventos

Cutscenes

Uma **cutscene** é uma sequência de quadros rotereizada em que o controle do jogador é temporariamente reduzido ou removido para transmitir a narrativa do jogo.

Super Mario Bros:
Cutscene de final de fase

Hollow Knight:
Cutscene de Introdução

Cutscenes Pré-Renderizadas

A forma mais simples de incluir cutscenes é utilizando vídeos (ex., mp4, wmv, mov) renderizados antecipadamente ou filmados e reproduzidos como um filme durante o jogo.

Screenshot do Adobe Premier:
Ferramenta de Edição de Vídeos

```
void Initialize() {
  ...
  M = LoadMovie("movie.mp4");
  ...
}

void Input() {
  ...
      PlayMovie(m);
  ...
}
```

Cutscenes Pré-Renderizadas

Pros

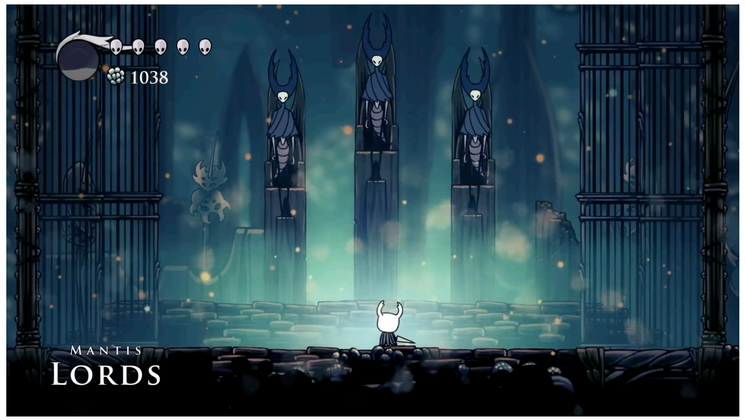
- Alta qualidade visual: pode ter uma aparência melhor do que gráficos em tempo real.
- Desempenho estável: não sobrecarrega a execução do jogo durante a reprodução.
- Liberdade de criação: possibilita uso de ferramentas externas de produção cinematográfica.
- Consistência entre plataformas: o vídeo tem a mesma aparência em todos os dispositivos.

Contras

- Tamanhos de arquivo grandes: vídeos de alta qualidade podem aumentar demais o tamanho do jogo.
- Baixa interatividade: não pode refletir as escolhas do jogador ou o estado do jogo.
- Quebra no estilo visual: pode parecer muito diferente dos gráficos do jogo.
- Desafios de localização: mais difícil de adicionar legenda ou dublar após a produção.
- Sem conteúdo dinâmico: não é possível reagir ao inventário, escolhas ou estatísticas.

Sistema de Cutscenes em Tempo Real

A maioria dos jogos modernos utilizam **cutsecenes renderizados em tempo real**. Sendo assim, a Engine deve dar suporte à criação de cenas roteirizadas. Os principais componentes:



1. Mecanismo de Acionamento (Trigger)
2. Sequenciador de Eventos:

- Controle de Câmera: movimentação, efeitos (zoom, shake, ...)
- Controle de Personagens: movimentação, animações
- Sistema de diálogo: legendas
- Sistema de som: música de fundo, efeitos sonros

Implementação de Cutscenes em Tempo Real

Existem três abordagens principais de implementação de cutscenes em tempo real , dependendo da complexidade do jogo e das funcionalidades/flexibilidade desejada:

1. **Cutscenes Hardcoded**

   Sequências escritas manualmente diretamente no código do jogo.

   Por exemplo, cutscene do Mario descendo o mastro no final das fases do SMB:

```
if (mIsOnPole) {
    // If Mario is on the pole, update the pole slide timer
    mPoleSlideTimer -= deltaTime;
    if (mPoleSlideTimer <= 0.0f) {
        mRigidBodyComponent->SetApplyGravity(true);
        mRigidBodyComponent->SetApplyFriction(false);
        mRigidBodyComponent->SetVelocity(Vector2::UnitX * 100.0f);
        mGame->SetGamePlayState(Game::GamePlayState::Leaving);

        mGame->GetAudio()->PlaySound("StageClear.wav");
        mIsOnPole = false;
        mIsRunning = true;
    }
}
```

Implementação de Cutscenes em Tempo Real

Existem três abordagens principais de implementação de cutscenes em tempo real, dependendo da complexidade do jogo e das funcionalidades/flexibilidade desejada:

2. **Interpretador de Cutscenes**

   As cutscenes são definidas em arquivos externos (JSON, XML, YAML, ...) e interpretadas por um sequenciador de eventos, que controla o fluxo de execução dos eventos.

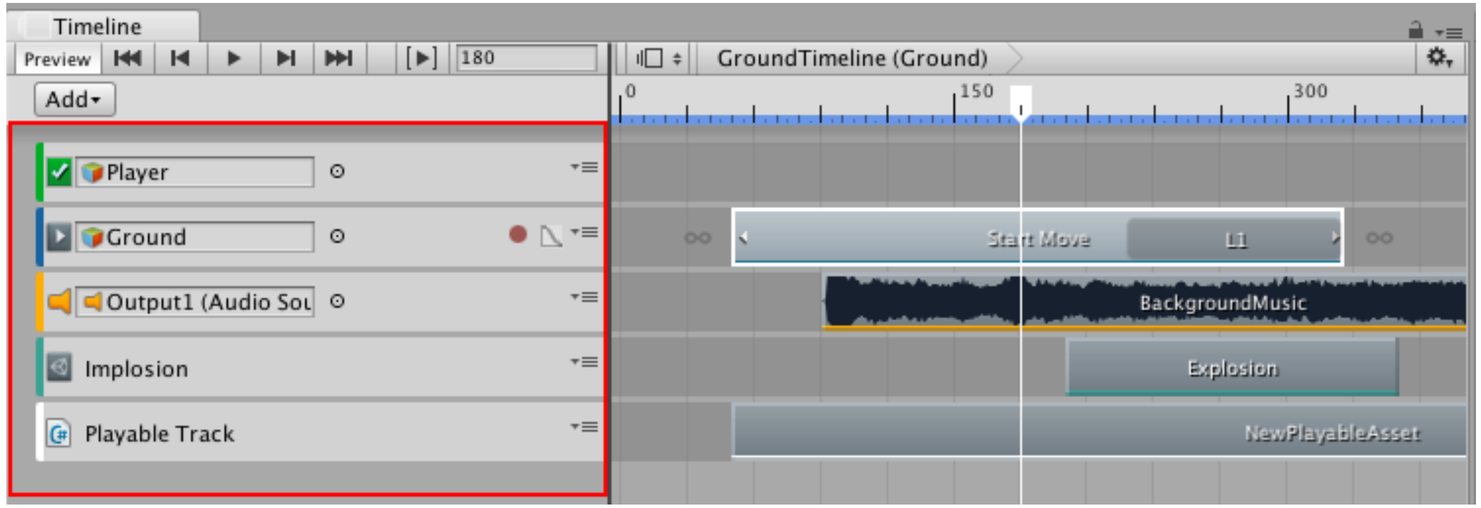
```
[
  {"event": "wait", "duration": 2},
  {"event": "move", "target": "npc1", "to": [5, 3], "duration": 2},
  {"event": "say", "target": "npc1", "text": "We're too late..."},
  {"event": "pan_camera", "to": "burned_house", "duration": 1},
  {"event": "wait", "duration": 2},
]
```

Implementação de Cutscenes em Tempo Real

Existem três abordagens principais de implementação de cutscenes em tempo real , dependendo da complexidade do jogo e das funcionalidades/flexibilidade desejada:

3. **Ferramentas de Sequênciamento Visual**

   As cutscenes são definidas por editor gráficos e interpretadas por um sequenciador de eventos, que controla o fluxo de execução dos eventos.



Screenshot do Unity Timeline: Subsistema de Cutscenes da Unity

Implementando um Interpretador de Cutscenes

Para implementar um **interpretador de cutscenes**, devemos primeiro definir como um cutscene é representada. A forma mais simples é utilizar uma lista de *eventos*:

1. Classe abstrata `CutsceneEvent` para especificar a estrutura mínima de um evento:

```
class CutsceneEvent
{
public:
    virtual ~CutsceneEvent() {}
    virtual void Start() = 0;
    virtual void Update(float deltaTime) = 0;
    virtual bool IsFinished() const = 0;

protected:
    float mDuration;
    float mElapsed;
};
```

Eventos possuem minimamente um tempo de duração

2. Especializamos a classe base para definir eventos específicos. Por exemplo:

```
class MoveEvent : public CutsceneEvent {
public:
    MoveEvent(Actor* obj, const Vector2
&destination, float seconds);

    void Start() override;
    void Update(float dt) override;
    bool IsFinished() const override;

Private:
    Actor* mTarget;
    Vector2 mStart, mEnd;
};
```

Implementando um Interpretador de Cutscenes

Após definir os eventos, podemos criar uma classe `CutScenePlayer` parar ler e tocar as cutsecenes definidas em arquivos:

```
class CutscenePlayer
{
public:
    void Start(const std::string& cutscene);
    void Update(float dt);
    void ReadCutscene(const std::string& fileName);
    bool IsPlaying() const { return mIsPlaying; }

private:
    size_t mCurrent = 0;
    bool mIsPlaying = false;

    // List of scheduled events
    std::vector<std::unique_ptr<CutsceneEvent>> mEvents;
};
```

- **`Start()`**: começa a tocar uma determinada cutscene
- **`Update()`**: gerencia início e término de eventos em uma cutscene.
- **`ReadCutscene()`**: lê arquivo texto estruturado (ex. json) definindo cutscenes
- **`IsPlaying()`**: Verifica se uma cutsene está tocando

Narrativas

Missões

Muitos jogos são estruturados em missões (*quests*), que são tarefas ou objetivos dados ao jogador para avançar na história, subir de nível, ou obter recompensas. Por exemplo:

Diablo II Resurrected - Ato 1 Quest 1: Den of Evil

Exemplos de missões:

- Matar todos os inimigos de uma área
- Coletar um item raro
- Salvar um personagem importante
- ...

Os atributos de uma missão

Uma missão possui, minimamente um **identificador único, um nome, uma descrição, uma lista de pré-requisitos (outras quests) e um estado**, que pode ser um dos seguintes:

1. **Não iniciada**
   A missão ainda não foi iniciada.

2. **Em execução**
   O jogador aceitou essa quest e está em progresso.

3. **Concluída**
   O jogador termiou essa quest com sucesso.

4. **Fracassada**
   O jogador termiou essa quest sem sucesso.

```cpp
enum class QuestState {
  NotStarted,
  InProgress,
  Completed,
  Failed
};

struct Quest {
    std::string id;
    std::string name;
    std::string description;
    QuestState state = QuestState::NotStarted;

   // Quest IDs
    std::vector<std::string> prerequisites;
};
```

Os métodos de uma missão

Uma missão também precisa ter métodos para verificar as condições de início (1) e fim (2) e de callbacks para atualizar o estado do jogo durante o seu início (3) e conclusão (4):

1. **Responder se ela pode ser iniciada**
   Ex.: Conversou com a vendedora de poções.

2. **Responder se ela já foi concluída**
   Ex.: Verificar se os monstros foram mortos.

3. **Modificar estado do jogo no início da missão**
   Ex.: Atualizar ponto de destino no mapa.

4. **Modificar estado do jogo no final da missão**
   Ex.: Dar uma recompensa para o jogador.

```
enum class QuestState {
  NotStarted,
  InProgress,
  Completed,
  Failed
};

struct Quest {
    std::string id;
    std::string name;
    std::string description;
    QuestState state = QuestState::NotStarted;

    std::vector<std::string> prerequisites;
    std::function<bool()> startCondition;
    std::function<bool()> completionCondition;
    std::function<void()> onStart;
    std::function<void()> onComplete;
};
```

Gerenciador de Missões

O gerenciador de missões é uma estrutura de dados tipicamente implementada como um mapa de quests indexadas por IDs (ex. strings).

```
private:
  // Lista de quests
  std::unordered_map<std::string, Quest> quests;
```

Essa estrutura deve verificar o estado de todas as quests do jogo:

```
public:
  bool AddQuest(const Quest& quest) const;
  const Quest* GetQuest(const std::string& id) const;
  void Update(float deltaTime);

private:
  bool ArePrerequisitesMet(const Quest& quest) const;
```

Gerenciador de Missões - Udpate

O método `update` verifica continuamente o estado das quests adicionadas:

- Se uma quest não foi começada, mas tem seus pre-requisitos e condição de início satisfeitos, ela será iniciada!
- Se uma quest está em progresso e tem sua condição de término satisfeita, ele será completada!
- Uma lógica similar pode ser usada para lidar com missões fracassadas

```
void update()
{
    for (auto& [id, quest] : quests)  {
        if (quest.state == QuestState::NotStarted && arePrerequisitesMet(quest) && quest.startCondition()) {
            quest.state = QuestState::InProgress;
            if (quest.onStart) quest.onStart();
        }

        if (quest.state == QuestState::InProgress && quest.completionCondition()) {
            quest.state = QuestState::Completed;
            if (quest.onComplete) quest.onComplete();
        }
    }
}
```

Gerenciador de Missões - Verificar Pré-requisitos

O método de verificação de pré-requisitos apenas percorre a lista de pré-requisitos de uma determinar quest e verifica se o estado de alguma delas não é concluído:

```
bool arePrerequisitesMet(const Quest& quest) const
{
    for (const auto& prereq : quest.prerequisites) {
        auto it = quests.find(prereq);
        if (it == quests.end() || it->second.state != QuestState::Completed)
            return false;
    }
    return true;
}
```

Exemplo

Vamos ver um exemplo de como uma criar uma quest simples usando esse sistema:

```
QuestManager questManager;

// Simulated game state variables
bool talkedToNPC = false;
bool defeatedBoss = false;

Quest quest {
        .id = "rescue_villager",
        .name = "Rescue the Villager",
        .description = "A villager is trapped in the dungeon. Talk to the guard to begin.",
        .startCondition = [&]() { return talkedToNPC; },
        .completionCondition = [&]() { return defeatedBoss; },
        .onStart = []() {
            std::cout << "Quest started: Rescue the Villager!\n";
        },
        .onComplete = []() {
            std::cout << "Quest completed: You rescued the villager and gained 500 XP.\n";
        }
};

questManager.addQuest(quest);
```

Sistema de Eventos

As condições de início e término de uma missão podem ser das mais variadas possíveis. Por exemplo, no Diablo II, quests são geralmente iniciadas quando o jogador:

- Entra em uma área;
- Mata uma chefe;
- Fala com um NPC;
- Pega um item;

Para facilitar a verificação dessas condições, você pode utilizar um sistema de eventos, que manda mensagens quando eventos acontecem!

Sistema de Eventos

O sistema de eventos dispara eventos, armazenados em uma lista, que podem ser utilizados pelo sistema de quest para verificar condições de início e fim:

```cpp
enum class GameEventType {
   EnterZone,
   KillEnemy,
   TalkToNPC,
   PickItem
};

struct GameEvent {
    GameEventType type;
    std::string data; // e.g., zone name, enemy id
};

std::vector<GameEvent> eventQueue;

// Example condition using event queue
auto bossDefeated = []() {
    return std::find_if(eventQueue.begin(), eventQueue.end(), [](const GameEvent& e) {
        return e.type == GameEventType::KillEnemy && e.data == "Andariel";
    }) != eventQueue.end();
};
```

Integração com NPCs

Para integrar o sistema de missões com os diálogos dos NPCs, podemos definir uma lista de missões relacionadas com cada NPC e tratar o diálogo caso a caso:

```
class NPC {
public:
    std::vector<std::string> relatedQuests;
    void onTalk(QuestManager& questManager) {
        for (const auto& questId : relatedQuests) {
            const Quest& quest = questManager->GetQuest(questId);
            switch (quest.state) {
                case QuestState::NotStarted:
                    std::cout << name << ": I have a task for you. Will you help?\n”;
                    eventQueue.push_back({ GameEventType::TalkToNPC, id });
                    return;
                case QuestState::InProgress:
                    std::cout << name << ": Have you finished what I asked?\n";
                    return;
                case QuestState::Completed:
                    std::cout << name << ": Thank you, brave hero!\n";
                    return;
                default:
                    break;
            }
        }
    }
};
```

Salvando o Jogo

Em jogos digitais, é comum que jogadores possam salvar o jogo, ou seja, armazenar seu progresso em memória externa, permitindo que ele retome o jogo de onde parou.

Você deve definir, enquanto designer, quais variáveis quer salvar:

- Nível do jogado, experiência, pontos, dinheiro, ...
- Posições no mundo
- Estado das missões
- Inventário
- ...

Você pode salvar o jogo:

- Em um arquivo texto estruturado (ex. json);
- Em um arquivo em formato binário (mais seguro!)
- Em um banco de dados! (principalmente em jogos online)

```json
{
  "player": {
    "name": "Hero123",
    "level": 12,
    "experience": 34560,
    "position": {
      "x": 100,
      "y": 250
    }
  },
  "quests": [
    {
      "id": "slay_skeleton_king",
      "state": "Completed"
    },
    {
      "id": "rescue_villager",
      "state": "InProgress"
    }
  ],
  "inventory": [
    {
      "item": "HealthPotion",
      "quantity": 5
    },
    {
      "item": "ManaPotion",
      "quantity": 3
    },
  ]
}
```

Próxima aula

A25: Jogos em Rede

- Protocolos de Rede
- Topologias de Rede
- Cheating