Manual de Boas Práticas e Condutas Obrigatórias

Projeto Unity – Uso Interno

---

0. Softwares Padronizados

Todos os membros da equipe devem utilizar as versões padronizadas dos softwares listados abaixo para garantir compatibilidade, consistência na integração dos recursos e evitar conflitos de versão.

Software	Versão mínima	Observações
Unity	2025.1 (Unity 6)	Versão oficial do projeto, com suporte LTS
Blender	4.2 LTS	Versão obrigatória para modelagem e animações
GIMP	2.10 ou superior	Alternativa para edição de imagens
Krita	5.2 ou superior	Alternativa para pintura digital e sprites
Audacity	2.4 ou superior	Utilizado para edição e tratamento de áudio

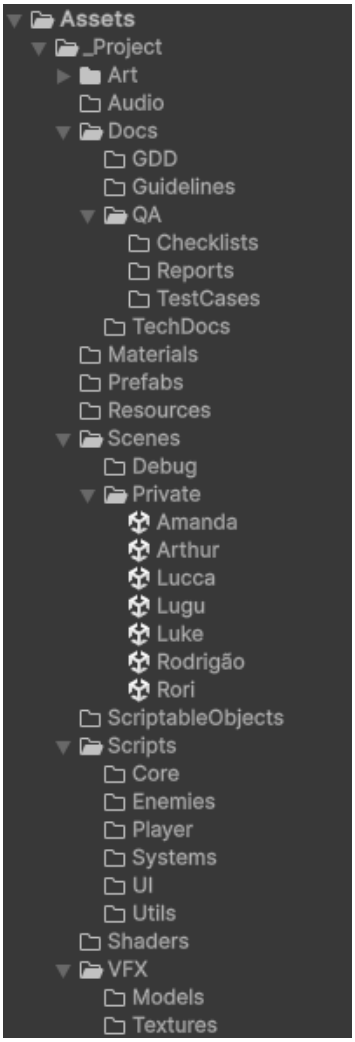
Regras Gerais:

- É proibido utilizar versões superiores não testadas sem autorização da liderança técnica.
- Todo asset criado ou editado deve ser compatível com a versão oficial adotada do Unity e seus respectivos pipelines.
- Caso haja necessidade de software alternativo, a substituição deve ser documentada e aprovada formalmente.

---

1. Estrutura de Pastas do Projeto

A organização de pastas do projeto é obrigatória e padronizada da seguinte forma. Toda a equipe deve seguir este modelo rigorosamente para garantir consistência, manutenção facilitada e integração fluida entre áreas.



Regras Gerais:

- Toda criação de arquivos deve respeitar a hierarquia acima.
- Evite criar pastas paralelas ou duplicadas fora dessa estrutura.
- Assets externos devem ser importados apenas em Plugins/ ou ThirdParty/.
- A pasta _Project/Art/Placeholder/ é destinada para arte temporária e protótipos, incluindo arquivos .blend em desenvolvimento.

- Arquivos .blend em desenvolvimento devem ser armazenados em _Project/Art/Placeholder/. Após a finalização, o arquivo .blend deve ser acompanhado do .fbx exportado correspondente na pasta definitiva, como Models/.

2. Convenções de Nomenclatura

2.1. Scripts e Código-fonte

- Scripts e nomes de classes: PascalCase; o nome do arquivo deve coincidir com o nome da classe pública.
- Métodos: PascalCase; preferencialmente verbo + complemento (MovePlayer(), ResetGame()).
- Variáveis públicas e propriedades: PascalCase; para exposição, prefira [SerializeField] private.
- Variáveis privadas: camelCase, com prefixo _ para campos (_currentHealth).
- Constantes: SCREAMING_SNAKE_CASE (MAX_HEALTH).
- Enums: PascalCase para enum e elementos.
- Eventos: On[Evento], use Action ou EventHandler.
- Interfaces: Começam com I + PascalCase (IDamageable).

2.2. Hierarquia da Cena e Objetos

- GameObjects: PascalCase, nomes descritivos e claros.
- Organizadores vazios: use sufixos como Group, Root ou Holder (UIRoot, EnemiesGroup).
- Prefabs: PascalCase com sufixos (_Prefab, _UI, _FX).

2.3. Assets do Projeto

- Sprites/Texturas: Categoria_Descrição sem espaços (Icon_Heart.png).
- Áudios: prefixos para categoria (SFX_, BGM_, VO_) + descrição.
- Animações: Objeto_Ação (Player_Idle.anim).
- Materiais: Objeto_Material (Player_Standard.mat).
- ScriptableObjects: prefixo SO_ + tipo + nome (SO_EnemyStats_Goblin.asset).

2.4. Tabela de Sufixos Obrigatórios

Sufixo	Aplicação	Exemplo
GO	GameObjects na hierarquia (em casos ambíguos)	PortalGO, LightTriggerGO
Group	Empty objects agrupadores	EnemiesGroup, UIGroup
Root	Objeto-pai raiz de um sistema	UIRoot, WorldRoot
Holder	Objeto vazio usado para armazenar elementos	AudioHolder, FXHolder
Ctrl	Controllers (lógica de controle)	PlayerCtrl, CameraCtrl
Manager	Classes ou objetos que gerenciam sistemas	GameManager, AudioManager
Spawner	Objetos ou scripts que instanciam prefabs	EnemySpawner, LootSpawner
SO_	Arquivos de ScriptableObject	SO_Item_HealthPotion
UI	Elementos de interface	MainMenuUI, ButtonUI
FX	Efeitos visuais	ExplosionFX, DashFX
SFX	Efeitos sonoros	SFX_Jump, SFX_Explosion
BGM	Trilha sonora de fundo	BGM_Level1, BGM_BossFight
VO	Voice-over (narrações e falas)	VO_Tutorial1, VO_FinalBoss
Anim	Controladores ou objetos de animação	PlayerAnim, EnemyAnim
State	Estados de máquina de estados	IdleState, AttackState
Data	Classes ou ScriptableObjects contendo dados puros	EnemyData, UIConfigData
Asset	Arquivos de dados ou recursos genéricos	DialogueAsset, LevelDataAsset
Config	Objetos de configuração geral	GameConfig, UIConfig
Event	Eventos customizados	GameOverEvent, OnItemPickedEvent

Input	Sistemas de entrada e controle de input	PlayerInput, MenuInput
Utils	Classes auxiliares ou estáticas	MathUtils, AudioUtils
Extension	Métodos de extensão	TransformExtension
Prefab	Arquivo de prefab	EnemyPrefab, ButtonUIPrefab
Mat	Material do Unity	PlayerMat, ButtonUIMat
Tex	Textura pura (não sprite)	GroundTex, SkyboxTex
Sprite	Sprite específico	HeartIconSprite, EnemyFaceSprite
Scene	Arquivos de cena	MainMenuScene, Level1Scene

---

3. Estruturação de Scripts

3.1. Organização e Responsabilidade de Classes

- Cada classe deve ter uma responsabilidade única e clara (Princípio da Responsabilidade Única).
- Scripts de MonoBehaviour devem ser focados em comportamentos relacionados a GameObjects.
- Lógica de dados deve ser separada em classes POCO ou ScriptableObjects.

3.2. Uso de MonoBehaviour, ScriptableObjects e Static

- Use MonoBehaviour para scripts que precisam estar vinculados a GameObjects.
- Use ScriptableObjects para dados configuráveis e compartilhados.
- Use classes estáticas para utilitários e funções globais sem estado.

3.3. Comentários e Documentação Interna

- Comente métodos públicos e complexos.
- Use XML comments para facilitar a geração automática de documentação.
- Evite comentários óbvios e mantenha o código autoexplicativo sempre que possível.

3.4. Reutilização e Escalabilidade

- Evite duplicação de código, use herança, interfaces e composição.
- Scripts devem ser modulares para facilitar testes e manutenções futuras.

3.5. Debugging, Logs, Padronização de Componentes e Organização do Código

- Use Debug.Log com categorias ou cores para diferenciar mensagens, mas sempre limpe ou desabilite logs em builds de produção para evitar impacto na performance e poluição no console.
- Crie wrappers para controle centralizado de logs, facilitando habilitar/desabilitar ou direcionar logs conforme o ambiente. Exemplo:
- Regras para uso de regions:
    - Use nomes claros e objetivos para as regions.

- Separe áreas distintas como variáveis públicas, privadas, propriedades, eventos, métodos Unity, métodos privados e utilitários.
- Não abuse do uso excessivo para evitar esconder código importante desnecessariamente.
- Prefira que o código esteja sempre autoexplicativo, regions são uma ajuda para organizar, não para esconder complexidade.

4. Boas Práticas de Programação

4.1. Uso de Eventos, Delegates e Actions

- Utilize eventos para comunicação desacoplada entre sistemas.
- Prefira Action e Func ao invés de delegates tradicionais, quando possível.

4.2. Uso de Corrotinas e Timers

- Corrotinas devem ser usadas para tarefas assíncronas e temporizadas.
- Evite corrotinas longas e complexas, divida em etapas.

4.3. Controle de Acesso e Encapsulamento

- Use modificadores de acesso para proteger dados (private, protected).
- Exponha apenas o necessário via propriedades públicas ou eventos.

4.4. Evitar Hardcoding e uso de Constantes

- Use ScriptableObjects ou arquivos de configuração para dados variáveis.
- Constantes e enums devem ser centralizados para facilitar ajustes.

5. Interface e UI

5.1. Organização de Hierarquia UI

- Use objetos vazios com sufixo Group para organizar elementos.
- Separe interfaces por tela ou funcionalidade.

5.2. Nomenclatura de elementos UI

- Use PascalCase com sufixo UI para objetos e scripts.
  Exemplo: MainMenuUI, ButtonStartUI.

5.3. Prefabs e Reutilização

- Prefabs de UI devem ser reutilizáveis e parametrizados.
- Evite duplicações na hierarquia.

5.4. Animações de UI

- Animadores devem ser nomeados com sufixo Anim.
- Use animações simples e otimizadas.

---

6. Arte e Animações

6.1. Modelagem 3D

- Software oficial: Blender 4.2 LTS.
- Modelos devem seguir a topologia limpa e otimizada, evitando polígonos desnecessários.
- Use unidades métricas (metros) para manter escala consistente com Unity.
- Nomes das malhas e objetos devem seguir nomenclatura PascalCase e conter prefixos/sufixos claros (ex.: Character_Hero, Prop_Tree).
- Sempre aplicar transformações (scale, rotation, position) antes de exportar.
- Exportar em formato .fbx com configurações compatíveis para Unity.
- Utilize UV mapping eficiente para texturização.
- Texturas devem estar dentro do padrão definido no projeto (png, tiff, etc.).
- Inclua LODs (níveis de detalhe) para objetos complexos sempre que possível.
- Sempre salve arquivos fonte .blend na pasta de origem.
- **Arquivos .blend em desenvolvimento devem ser colocados na pasta Art/Placeholder/. Após finalizados, o arquivo .blend deve acompanhar seu .fbx correspondente na pasta definitiva.**

6.2. Texturização e Material

- Utilize ferramentas como GIMP 2.10+, Krita 5.2+ para criação e edição de texturas.
- Use naming convention consistente: Objeto_Tipo.extensão (Tree_Diffuse.png, Player_Normal.png).
- Prefira texturas de resolução balanceada para performance e qualidade.
- Separe texturas em mapas: Difuso, Normal, Metallic, Roughness, etc.
- Materiais no Unity devem ser nomeados com o sufixo Mat e organizados na pasta Materials/.

6.x. Formatos de Arquivos Ideais

- **Texturas e Sprites**

    - Formato principal: **PNG** (suporta transparência, compressão sem perdas, ideal para sprites e UI)
    - Outras opções permitidas: **TGA** (quando precisar de canais alpha), **JPEG** (para texturas sem transparência e fundo contínuo, evitando tamanho muito grande)
    - **Resolução obrigatória: potências de 2 com tamanho mínimo de 512x512 pixels.**
    - Texturas maiores que 512x512 só podem ser usadas após aprovação da gerência do projeto, para evitar impactos na performance e tamanho do build.
    - Ícones de UI e sprites devem seguir essa regra e ser otimizados para balancear qualidade e desempenho.
- **Modelos 3D**
    - Formato oficial: **FBX** (compatível nativo com Unity, suporta animação, materiais e rigging). Observação: Cada artista é responsável pela limpeza dos próprios FBX, a não consistência nessa prática pode acarretar em reports por parte da gerência.
    - Arquivos fonte: **.blend** (Blender) mantidos fora da pasta Assets para edição e versionamento.
- **Áudio**
    - Formato principal para efeitos sonoros: **WAV** sem compressão para qualidade máxima em desenvolvimento.
    - Para builds, utilize **OGG** para compressão com qualidade balanceada e tamanho reduzido.
    - Música de fundo (BGM): preferencialmente OGG ou MP3, balanceando qualidade e tamanho.
- **Documentos**
    - Use **PDF** para documentação finalizada e **DOCX** para documentos em edição colaborativa.

6.3. VFX (Efeitos Visuais)

- Prefira criar efeitos utilizando o sistema oficial de partículas do Unity ou ferramentas aprovadas.
- Nomes de prefabs de efeitos devem conter o sufixo FX (ex: ExplosionFX).

- Mantenha efeitos leves para não comprometer performance, evite partículas excessivas.
- Organize efeitos em pastas específicas dentro de Prefabs/FX/ ou Resources/FX/.
- Scripts de controle de efeitos devem estar na pasta Scripts/Systems/FX/.
- Use sistemas modulares que possam ser reutilizados e ajustados rapidamente.
- Documente o funcionamento e parâmetros dos efeitos mais complexos.

6.4. Animações

- Use o sistema de animação do Unity (Animator, Animation Clips).
- Nomeie animações com o padrão Objeto_Ação (Player_Run, Enemy_Attack).
- Controle de animações com controladores nomeados com sufixo Anim (PlayerAnim).
- Utilize blend trees para transições suaves.
- Organize animações em pastas específicas, separadas por personagem ou tipo.
- Inclua documentação sobre regras de transição e condições.

6.5. Padrões de Importação

- Ajuste configurações de importação para cada asset (ex: compressão, escala, rigging).
- Use presets padronizados sempre que possível para garantir uniformidade.
- Evite reimportações desnecessárias que possam causar conflitos ou perdas.

7. Áudio

7.1. Padrões de Nomenclatura

- Use prefixos SFX_, BGM_, VO_ conforme aplicável.

7.2. Compressão e Configurações

- Ajuste configurações para balancear qualidade e performance.

7.3. Categorias e Controle de Volume

- Gerencie volumes via AudioManager centralizado.

7.4. Integração com AudioManager

- Use eventos e sistemas de mixagem para controle dinâmico.

8. Level Design

8.1. Criação e Organização de Cenas

- Nomeie cenas com padrão NomeCenaScene.
- Use pastas dedicadas para cenas relacionadas.

8.2. Reutilização de Prefabs e Modularidade

- Prefabs modulares para facilitar edição e reutilização.

8.3. Regras para Triggers, Interações e Checkpoints

- Siga nomenclatura e padrões para triggers e volumes.

8.4. Padronização de Layers e Tags

- Use layers e tags definidas e documentadas.

---

9. Workflow de Equipe e Versionamento

O projeto utiliza GitHub como ferramenta oficial para versionamento, gerenciamento de tarefas, integração de código e revisão entre membros da equipe. Toda a colaboração e entrega de trabalho deve passar por esse fluxo.

9.1. Uso do GitHub

- Todo código deve estar versionado no repositório oficial do projeto no GitHub.
- O trabalho deve ser realizado em branches separados, nunca diretamente na branch principal.
- É obrigatório usar commits claros e descritivos.
- Issues, pull requests, milestones e boards são utilizados para acompanhar tarefas e progresso.

9.2. Nome de Branches

- As branches devem seguir o padrão:
  tipo/ID-descricao-curta
- Onde tipo pode ser:
  - feature
  - bugfix
  - hotfix
  - refactor
  - test
- Exemplos:
  - feature/32-inventario-basico
  - bugfix/45-colisao-inimigo
  - hotfix/70-ui-bloqueada

9.3. Pull Requests e Reviews

- Todo código deve ser integrado à branch principal via Pull Request (PR).

- PRs devem ser vinculados a uma issue correspondente e conter descrição clara do que foi implementado.
- É obrigatório solicitar code review (ainda válido para tarefas relacionadas à arte) de pelo menos dois membros da equipe, sendo um deles obrigatoriamente da gerência da sua área de atuação.
- Revisores devem:
    - Verificar se o código segue as boas práticas do projeto;
    - Validar se a funcionalidade está correta;
    - Sugerir melhorias ou ajustes necessários.

9.4. Commits

- Commits devem ser pequenos, frequentes e atômicos.
- A mensagem de commit deve seguir o padrão:
  [type] ID - descrição breve
- Onde type pode ser: feat, fix, refactor, style, docs, test.
- Exemplos:
    - feat 32 - Adiciona lógica de drop de itens
    - fix 45 - Corrige colisão com objetos dinâmicos
    - docs - Atualiza documentação do sistema de inventário

9.5. Tarefas e Organização

- Atribuição de tarefas é feita via GitHub com responsáveis definidos.
- Cada tarefa deve conter:
    - Descrição clara do escopo;
    - Critérios de aceite;
    - Checklists ou requisitos técnicos, se necessário.

9.6. Integração e Conflitos

- Sempre sincronize com a branch principal antes de abrir um PR.
- Conflitos devem ser resolvidos localmente e com cuidado, preferencialmente em pares.
- Evite commits diretamente na branch principal.
- **9.7. Regra Especial para Commits em Branchs Coletivas**

- Commits em branches coletivas (ex: develop, release) **só podem ser realizados pela gerência do projeto.**
- Desenvolvedores devem fazer PRs para essas branches, que serão revisadas e integradas pela gerência.

- **10. Documentação e Testes**
- **10.1. Documentação**
- Toda documentação deve estar armazenada na pasta Assets/_Project/Docs/.
- Divida a documentação entre:
- GDD (Game Design Document)
- TechDocs (arquitetura e notas técnicas)
- Guidelines (boas práticas e padrões)
- QA (teste) com casos de teste, relatórios e checklists
- **10.2. Testes**
- Crie casos de teste para funcionalidades críticas.
- Documente os passos, dados e resultados esperados.
- Use relatórios para acompanhamento de bugs e melhorias.
- Mantenha checklists de verificação para validações regulares.