UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

FELIPE DA SILVA FADEL

UMA ABORDAGEM PARA VERSIONAMENTO DE ARQUIVOS BINÁRIOS BASEADA EM ARMAZENAMENTO DIFERENCIAL POR BLOCOS

GUARAPUAVA

2026

FELIPE DA SILVA FADEL

UMA ABORDAGEM PARA VERSIONAMENTO DE ARQUIVOS BINÁRIOS BASEADA EM ARMAZENAMENTO DIFERENCIAL POR BLOCOS

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Sistemas para Internet pela Universidade Tecnológica Federal do Paraná (UTFPR).

Orientadora: Profª Drª Sediane Carmem Lunardi Hernandes

Coorientador: Prof. Dr Andres Jessé Porfirio

GUARAPUAVA
2026

4.0 Internacional

Esta licença permite compartilhamento, remixe, adaptação e criação a partir do trabalho, mesmo para fins comerciais, desde que sejam atribuídos créditos ao(s) autor(es). Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

RESUMO

O desenvolvimento colaborativo de software, especialmente na indústria de jogos digitais, envolve o uso intensivo de arquivos binários de grande porte, como modelos tridimensionais, texturas e áudios. Os sistemas tradicionais de controle de versão, ferramentas que registram e gerenciam o histórico de alterações em arquivos ao longo do tempo, não lidam de forma eficiente com esse tipo de arquivo, pois armazenam cópias completas a cada modificação, mesmo quando apenas pequenas partes do conteúdo foram alteradas. Soluções existentes no mercado atenuam parcialmente esse problema, mas ainda tratam os arquivos binários como blocos indivisíveis, sem analisar seu conteúdo interno. Este trabalho propõe uma abordagem de versionamento que armazena apenas as partes efetivamente modificadas de um arquivo binário tridimensional, aproveitando sua estrutura interna para identificar e isolar cada segmento de dados. Para detectar alterações, aplica-se uma função de SHA-256 sobre cada segmento, gerando uma assinatura digital única que permite comparar versões com precisão. O armazenamento diferencial é realizado por meio de um sistema de arquivos moderno, como as Btrfs que, ao registrar uma modificação, preserva os dados anteriores sem sobrescrevê-los, compartilhando os blocos inalterados entre as versões. Os experimentos compararam o método proposto com a abordagem convencional de armazenamento de arquivos grandes em repositórios de código, em cenários de modificação progressiva de um modelo tridimensional. Os resultados iniciais indicam reduções expressivas no espaço consumido, chegando a economias de até 98% por versão nos cenários com poucas alterações, e de aproximadamente 39% no espaço total acumulado ao longo de múltiplas versões.

Palavras-chave: versionamento de arquivos binários; armazenamento diferencial; preservação de blocos; sistemas de arquivos modernos.

ABSTRACT

Collaborative software development, especially in the digital game industry, relies heavily on large binary files such as three-dimensional models, textures, and audio assets. Traditional version control systems, which are tools designed to track and manage file changes over time, handle these files inefficiently by storing complete copies on every modification, even when only small portions of the content have changed. Existing solutions partially address this issue but still treat binary files as indivisible units, without examining their internal structure. This work proposes a versioning approach that stores only the parts of a three-dimensional binary file that were effectively modified, leveraging its internal structure to identify and isolate each data segment. To detect changes, the SHA-256 function is applied to each segment, generating a unique digital signature that enables precise version comparison. Differential storage is performed through a modern file system, such as Btrfs, which upon recording a modification preserves previous data without overwriting it, sharing unchanged blocks across versions. Experiments compared the proposed method against the conventional approach of storing large files in code repositories, across progressive modification scenarios applied to a three-dimensional model. Initial results indicate expressive reductions in storage consumption, reaching savings of up to 98% per version in low-change scenarios, and approximately 39% in total accumulated storage across multiple versions.

Palavras-chave: Keywords: binary file versioning; differential storage; block preservation; modern file systems.

LISTA DE ABREVIATURAS E SIGLAS

Siglas

ASCII	*American Standard Code for Information Interchange*
BIN	*Binary File*
Btrfs	*B-tree File System*
CoW	*Copy-on-Write*
DAG	*Directed Acyclic Graph*
GLB	*GL Transmission Format Binary*
glTF	*GL Transmission Format*
Git LFS	*Git Large File Storage*
HEAD	Referência para o *commit* atual no Git
JSON	*JavaScript Object Notation*
LFS	*Large File Storage*
PBR	*Physically Based Rendering*
RoW	*Redirect-on-Write*
SHA-256	*Secure Hash Algorithm 256 bits*
URI	*Uniform Resource Identifier*
UTF-8	*Unicode Transformation Format 8-bit*
UTFPR	Universidade Tecnológica Federal do Paraná
VM	*Virtual Machine*
WebGL	*Web Graphics Library*
ZFS	*Zettabyte File System*

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de *software* se torna mais complexo a cada dia e, com o surgimento constante de novas tecnologias e ferramentas, a quantidade de arquivos, métodos e estruturas envolvidos nos projetos cresce significativamente. Segundo Pressman e Maxim (2016), essa complexidade crescente exige práticas cada vez mais organizadas para garantir qualidade, manutenibilidade e colaboração eficiente ao longo do ciclo de vida do *software*.

Na indústria de desenvolvimento de jogos digitais, essa complexidade é ainda mais evidente, pois esse setor reúne programação e entretenimento. Criar um jogo é um trabalho que envolve diversos profissionais de diferentes áreas e, analogamente ao que afirma Schell (2008), pode-se destacar a atuação dos programadores, responsáveis por transformar código em mecânicas, e dos artistas sonoros e visuais, que dão vida a essas mecânicas. Por exemplo, o programador cria o personagem que corre, o artista visual a animação e o artista sonoro o som da corrida. Por consequência, o desenvolvimento de jogos caracteriza-se como um processo essencialmente colaborativo, no qual múltiplas equipes trabalham simultaneamente sobre os mesmos arquivos e sistemas.

Nesse sentido, quando se observa o cenário de colaboração, torna-se necessário o uso de ferramentas que permitam o trabalho conjunto de forma organizada e segura. Nesse contexto, destaca-se o Git, que, segundo Cunha (2018), é um sistema de controle de versões distribuído que permite a colaboração recíproca entre desenvolvedores, no qual cada integrante trabalha em seu ambiente local, ou repositório, e, após concluir uma tarefa, envia suas alterações para o repositório principal do projeto.

Entretanto, apesar de sua ampla utilização e eficiência no controle de versões de arquivos textuais, como código-fonte, o Git apresenta limitações no gerenciamento de arquivos binários de grande porte, comuns no desenvolvimento de jogos, como imagens, modelos tridimensionais, áudios e vídeos. Para mitigar esse problema, utiliza-se o *Git Large File Storage* (Git LFS), uma extensão que armazena apenas o endereçamento desses arquivos no repositório principal, mantendo os dados reais em um servidor externo. Entretanto, essa abordagem não resolve plenamente os desafios relacionados ao versionamento eficiente, ao consumo de armazenamento e ao controle detalhado das alterações em arquivos binários.

Diante desse contexto, observa-se que os mecanismos tradicionais de controle de versões apresentam limitações significativas no gerenciamento eficiente de arquivos binários de grande porte. Embora soluções como o Git LFS reduzam parcialmente o impacto desses arquivos nos repositórios, tais abordagens ainda tratam os dados binários como unidades

indivisíveis, resultando no armazenamento completo de novas versões mesmo quando apenas pequenas partes do arquivo foram modificadas. Nesse cenário, surge a necessidade de investigar abordagens alternativas capazes de lidar de forma mais eficiente com o versionamento desses dados e analisar também quais os melhores dados para tal.

Desta forma, este trabalho busca responder à seguinte questão de pesquisa: Como técnicas de armazenamento diferencial por blocos, baseadas nos princípios de *Copy-on-Write* e *Redirect-on-Write*, podem ser aplicadas ao versionamento de arquivos binários para reduzir custos de armazenamento e melhorar o desempenho em ambientes colaborativos de desenvolvimento de software?

1.1 Justificativa

Ao tratar de arquivos binários no contexto de ambientes de produção, é possível citar exemplos como arquivos de áudio, imagens em alta resolução, modelos tridimensionais e até mesmo modelos de inteligência artificial. Esses arquivos, quando armazenados em repositórios que seguem fluxos tradicionais de versionamento, os quais mantêm o histórico completo das alterações, apresentam limitações significativas. Tal cenário ocorre devido à impossibilidade de comparar esses dados de forma eficiente por meio de operações de diferença, *diff*, além da dificuldade de compactá-los adequadamente, o que pode comprometer o desempenho e o armazenamento do repositório ao longo do tempo.

Git LFS surge como uma solução paliativa para esse problema, pois substitui arquivos binários de grande porte por arquivos de ponteiro que referenciam servidores externos, nos quais os dados reais são armazenados (Chen et al., 2025). Embora essa abordagem reduza o tamanho do repositório principal, ela não resolve de maneira estrutural os desafios relacionados ao versionamento eficiente desses arquivos.

O principal problema reside no fato de que os arquivos binários passam a ser tratados como “caixas pretas”, cujo conteúdo interno não pode ser analisado e comparado. Por consequência, qualquer modificação, mesmo que mínima, resulta no envio e armazenamento de uma nova versão completa do arquivo, ocasionando desperdício significativo de espaço em disco, aumento no tráfego de dados e impacto negativo no desempenho dos sistemas de versionamento.

Diante desse cenário, torna-se necessário investigar alternativas capazes de evitar a substituição integral desses binários, permitindo o registro apenas das porções efetivamente modificadas dos arquivos. Nesse contexto, propõe-se a utilização de sistemas de

versionamento e armazenamento fundamentados nos princípios de *Copy-on-Write* (CoW) e *Redirect-on-Write* (RoW), amplamente utilizadas em sistemas de armazenamento para a criação de *snapshots*[1] diferenciais, conforme apresentado por Xiao (2009). Essas abordagens possibilitam a preservação de versões anteriores por meio do armazenamento seletivo de blocos alterados, reduzindo significativamente o consumo de espaço e os custos computacionais associados às operações de escrita.

Nesse sentido, a aplicação desses conceitos ao contexto do versionamento de arquivos binários apresenta-se como uma alternativa promissora para superar as limitações dos mecanismos tradicionais, possibilitando maior eficiência no armazenamento, melhoria no desempenho e controle mais refinado das alterações, especialmente em ambientes colaborativos de grande escala, como os encontrados na indústria de desenvolvimento de jogos digitais, onde os arquivos binários podem mudar a mesma medida que os arquivos de texto.

Além das limitações relacionadas ao armazenamento e ao desempenho, estudos recentes evidenciam fragilidades significativas quanto à integridade e à segurança do Git LFS. Conforme apresentado por Chen et al. (2025), foram identificadas vulnerabilidades que permitem a substituição de arquivos legítimos por conteúdo malicioso, bem como o vazamento de arquivos confidenciais, em virtude da ausência de avaliações rigorosas de integridade durante o processo de transferência e recuperação dos dados.

Ainda segundo Chen et al. (2025), foram identificadas 36 vulnerabilidades distribuídas em 14 grandes plataformas de hospedagem *Git*, sendo que 85,7% delas violaram ao menos uma das propriedades de segurança definidas no artigo. As falhas mais recorrentes incluem evasão de cota, em que os usuários conseguem burlar o armazenamento e colocar arquivos mais pesados, ataques de negação de serviço baseados em cota, substituição de arquivos e vazamento de dados, evidenciando limitações estruturais significativas no modelo atual de gerenciamento de arquivos binários.

2 OBJETIVOS

Os objetivos do trabalho se dividem em geral e específicos.

[1] Snapshot é uma técnica de versionamento utilizada em sistemas de backup que cria uma imagem do estado do sistema de arquivos em um determinado momento (Xiao, 2009).

2.1 Objetivo geral

Desenvolver um novo mecanismo de versionamento de arquivos binários fundamentado em arquiteturas baseadas nos princípios de *Copy-on-Write* (CoW) e *Redirect-on-Write* (RoW), visando otimizar o armazenamento, o desempenho e o controle das alterações, especialmente no contexto do desenvolvimento de jogos digitais em três dimensões.

2.2 Objetivos específicos

- Analisar as limitações dos mecanismos tradicionais de versionamento no tratamento de arquivos binários, incluindo a identificação de cenários nos quais a divisão em blocos seja viável como objeto de estudo.
- Investigar a aplicação das técnicas *Copy-on-Write* (CoW) e *Redirect-on-Write* (RoW) no contexto de armazenamento diferencial por blocos, considerando sistemas de arquivos como o *B-tree File System* (Btrfs) e o *Zettabyte File System* (ZFS), bem como seu funcionamento no Kernel do Linux e sua aplicabilidade ao problema proposto.
- Propor uma abordagem para o versionamento de arquivos binários baseada em armazenamento diferencial por blocos, avaliando sua eficiência em termos de consumo de armazenamento e desempenho, por meio de análise comparativa com abordagens utilizadas no mercado.

3 REFERENCIAL TEÓRICO

A seguir é apresentado o referencial teórico que fundamenta o desenvolvimento de uma abordagem de versionamento. Inicialmente, são discutidos os conceitos relacionados aos sistemas de controle de versões, com ênfase no funcionamento e nas características do sistema Git e de suas extensões voltadas ao gerenciamento de arquivos de grande porte. Em seguida, são apresentados diferentes modelos de escrita utilizados em sistemas de arquivos modernos, abordando as estratégias empregadas para registrar modificações em dados e preservar versões anteriores. Posteriormente, são discutidos sistemas de arquivos que utilizam essas abordagens, bem como aspectos relacionados ao tratamento de arquivos binários em processos de armazenamento e versionamento. Por fim, é apresentado o objeto de teste

adotado neste trabalho, um tipo de arquivo binário utilizado na modelagem tridimensional que apresenta características relevantes para os processos de versionamento, sendo descrita sua estrutura e funcionamento.

3.1 Sistema de controle de versões

Um sistema de controle de versões é uma ferramenta utilizada durante o fluxo de trabalho de desenvolvimento para gerenciar alterações em arquivos ao longo do tempo. Esse controle permite rastrear o histórico de evolução de um projeto e possibilita a colaboração entre múltiplos desenvolvedores simultaneamente. Por meio de mecanismos como registro de alterações, revisão de código, gerenciamento de versões e ferramentas de colaboração, esses sistemas organizam o desenvolvimento de forma estruturada, garantindo a integridade do código e a coordenação entre diferentes contribuições, além de permitir o gerenciamento das diferentes versões de um sistema ou aplicação (SPINELLIS, 2005).

O uso de sistemas de controle de versão é uma prática fundamental para o desenvolvimento profissional de *software*; pode-se comparar a prática de usá-los com a importância do uso de editores de texto ou compiladores. Implementar o controle de versões a nível empresarial pode envolver custos iniciais e uma curva de aprendizado, mas a sua ausência compromete a organização e a evolução de projetos de produção (SPINELLIS, 2005).

Um sistema de controle de versões atua, na prática, como uma máquina do tempo para cada projeto, permitindo que a equipe de desenvolvimento gerencie o histórico de alterações de forma ampla e segura. Em cenários de trabalho em equipe, a principal vantagem é evitar que os desenvolvedores sobrescrevam o trabalho alheio (SPINELLIS, 2005). Como exemplos de sistemas de controle de versão tem-se o Subversion e o Git.

3.1.1 O Sistema de Controle de Versão Git

O Git é um sistema de controle de versão distribuído amplamente adotado devido à sua rapidez e eficiência no gerenciamento de projetos de *software*, em contraste com os modelos centralizados. O Git permite que cada colaborador possua um repositório privado completo em seu ambiente de trabalho. Essa arquitetura descentralizada garante que os desenvolvedores possam realizar *commits*, editar históricos e criar ramificações de forma

independente, sem afetar o trabalho de terceiros ou depender de uma conexão constante com o servidor principal (CUNHA, 2018).

No Git, o estado atual do trabalho é monitorado por um ponteiro chamado *HEAD*, que indica a ramificação e o comando mais recente realizado. Todas as ações executadas localmente são registradas automaticamente em um arquivo de registro, que funciona como um rastro histórico das ações realizadas no repositório. Para consolidar o trabalho individual no repositório principal, os desenvolvedores utilizam mecanismos de integração, sendo o *merge* o mais tradicional por registrar explicitamente a união de duas linhagens de código em um novo *commit* (CUNHA, 2018).

Apesar de sua eficiência em relação ao código-fonte, o Git enfrenta gargalos significativos ao lidar com conteúdo binário e arquivos de grande porte, uma vez que o sistema foi projetado para armazenar o histórico completo de cada arquivo. Contudo, arquivos binários não permitem operações de comparação diferencial ou compressão de forma eficaz, como ocorre em arquivos de texto. Consequentemente, qualquer pequena alteração em um binário resulta no armazenamento de uma nova cópia integral do arquivo, causando uma rápida e ineficiente expansão do tamanho do repositório. Essa limitação motiva o uso de extensões e de abordagens alternativas de armazenamento que operem de forma mais granular (CHEN et al., 2025).

Entre essas soluções, tem-se o Git LFS, uma extensão amplamente adotada na indústria de software para lidar com arquivos pesados, que, ao serem versionados, normalmente vão inflar o armazenamento do repositório tradicional do Git (CHEN et al., 2025). A base de funcionamento do Git LFS se baseia na substituição de arquivos binários relativamente grandes por arquivos de ponteiro textuais dentro do repositório principal. Esses ponteiros servem como referências para o conteúdo real do arquivo que está em um ambiente externo, um servidor LFS especializado. Com isso, o repositório Git mantém apenas algumas informações sobre o assunto, como o identificador criptográfico, *hash*, e o tamanho do arquivo. Os dados do binário são recuperados apenas durante operações de sincronização ou recuperação de versões do repositório.

Essa abordagem permite reduzir significativamente o crescimento do repositório principal e otimizar processos como clonagem e sincronização de projetos. No entanto, ao deslocar o armazenamento dos arquivos para uma infraestrutura externa, o Git LFS introduz novas camadas de complexidade relacionadas à gestão de recursos, ao desempenho da infraestrutura e à segurança do sistema (CHEN et al., 2025).

Estudos recentes apontam que a arquitetura baseada em servidores externos pode ampliar a superfície de ataque do sistema, especialmente quando os mecanismos de verificação de integridade e controle de acesso não são implementados de forma rigorosa. Um exemplo conhecido dessas limitações ocorreu na plataforma GitLab, por meio da vulnerabilidade identificada como CVE-2019-6786, na qual falhas nos mecanismos de validação permitiam que usuários mal-intencionados acessassem arquivos privados armazenados em ambiente externo. Nesse caso, controles inadequados de acesso permitem a exploração do sistema por meio do envio de objetos vazios, contornando verificações de autorização e recuperando conteúdos que deveriam permanecer restritos (Chen et al., 2025).

3.2 Modelos de escrita para sistemas de arquivos

Existem duas políticas fundamentais utilizadas para atualizar blocos de dados em dispositivos de armazenamento: o modelo de sobrescrita direta, conhecido como *Update In Place* (UIP), e abordagens que evitam a modificação direta do conteúdo previamente gravado. No modelo UIP, o bloco de dados é inicialmente carregado para a memória, modificado e posteriormente gravado novamente no disco exatamente na mesma posição física, substituindo o conteúdo anterior. Em contrapartida, arquiteturas mais recentes adotam estratégias nas quais os dados modificados são registrados em novas regiões do dispositivo de armazenamento, preservando a versão anterior do bloco (CHEN et al., 2014).

Essas abordagens de escrita são amplamente utilizadas em diferentes áreas da computação. Além de sistemas de arquivos modernos, neles os mecanismos também são empregados em sistemas de gerenciamento de bancos de dados para controle transacional, na gerência de memória virtual para compartilhamento eficiente de páginas entre processos e em ambientes de virtualização utilizados na criação de discos virtuais. O objetivo principal dessas estratégias é aumentar a confiabilidade do sistema e facilitar a manutenção da consistência dos dados, permitindo a recuperação em situações de falha e viabilizando recursos como snapshots e clonagem de estados do sistema. Nas subseções seguintes são apresentados os principais modelos utilizados para implementar essas estratégias de atualização de dados (XIAO et al., 2009), as quais são utilizadas nos sistemas de arquivos modernos, como *Btrfs* e *ZFS*.

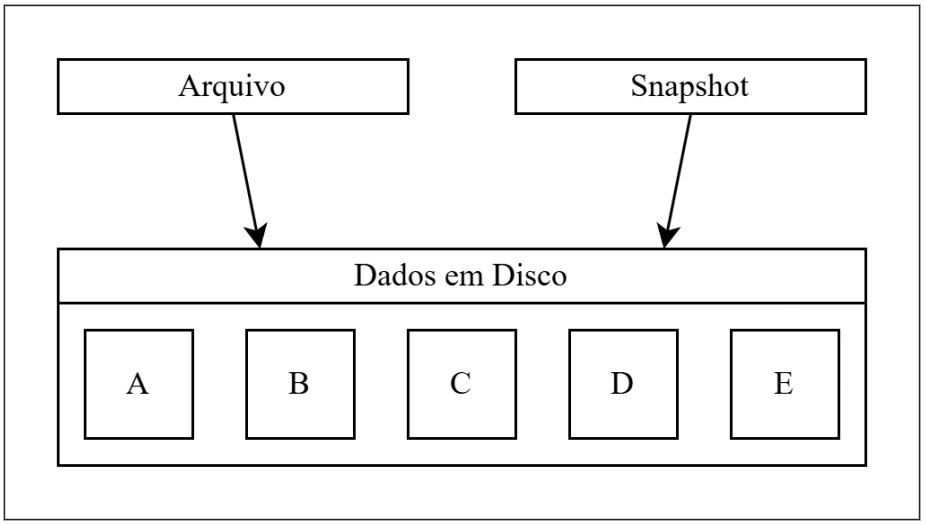
3.2.1 *Copy-On-Write* e *Redirect-On-Write*

O modelo *Copy-on-Write* (CoW) possui como princípio fundamental evitar a sobrescrita direta de dados já existentes, garantindo que versões anteriores possam ser preservadas. Enquanto múltiplos processos necessitam apenas realizar operações de leitura sobre um determinado dado, eles podem compartilhar ponteiros para a mesma estrutura em disco. Entretanto, quando ocorre a primeira tentativa de escrita em um bloco após a criação de um *snapshot*, o sistema executa uma sequência de operações: inicialmente, o bloco original é lido; em seguida, uma cópia desse bloco é registrada no volume associado ao *snapshot*; por fim, a nova versão modificada do bloco é gravada na estrutura ativa do sistema de arquivos (XIAO et al., 2009).

Esse mecanismo pode ser comparado, de forma análoga, à aquisição de um imóvel: embora exista um custo inicial mais elevado associado à sua obtenção, as operações subsequentes de uso e manutenção tornam-se mais diretas e eficientes. De maneira semelhante, o CoW introduz um custo adicional no momento da primeira modificação de um bloco, mas permite otimizações significativas nas operações posteriores (PETERSON; BURNS, 2005).

Para ilustrar o funcionamento desse modelo, considera-se um arquivo binário dividido em cinco blocos de dados. Na Figura 1, observa-se o estado inicial do arquivo, no qual cada bloco é referenciado por ponteiros armazenados no sistema de arquivos (XIAO et al., 2009).

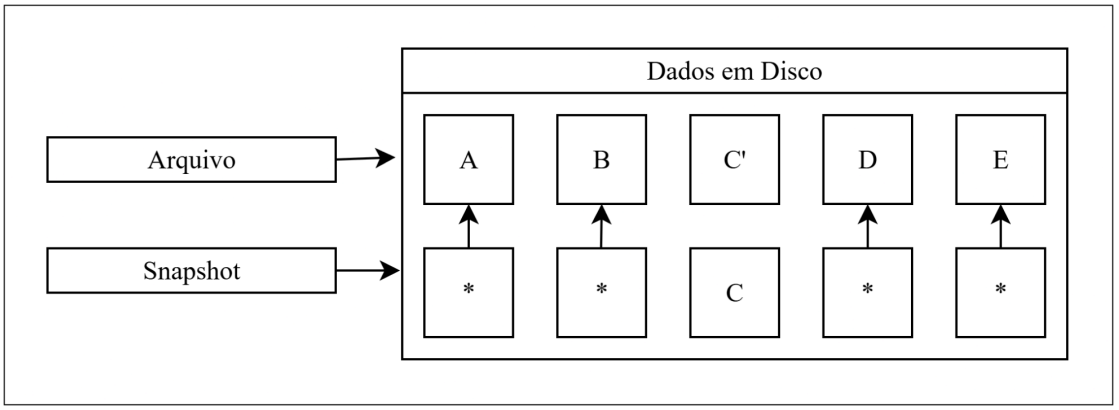
Figura 1 - Esquema de ponteiros no disco.



Fonte: Autoria própria (2026).

Conforme demonstrado na Figura 2, quando ocorre uma modificação em um dos blocos do arquivo, o mecanismo CoW evita a sobrescrita direta do bloco original. Em vez disso, um novo bloco é alocado no disco para armazenar a versão modificada dos dados, enquanto o bloco original permanece preservado. Os ponteiros do sistema são então atualizados para referenciar o novo bloco, mantendo intacta a versão anterior dos dados. Olhando o snapshot ilustrado na Figura 2, observa-se que apenas o bloco C foi modificado após a criação da imagem do sistema. Como resultado, o mecanismo CoW cria um novo bloco C′, para armazenar a versão atualizada dos dados, enquanto o bloco C original permanece preservado e continua sendo referenciado pelo snapshot. Os demais blocos (A, B, D e E) não sofreram alterações e, portanto, permanecem compartilhados entre o estado atual do arquivo e o snapshot (XIAO et al., 2009).

Figura 2 - Disco após as mudanças no arquivo.



Fonte: Autoria própria (2026).

Esse comportamento permite a criação eficiente de versões e *snapshots* do arquivo, uma vez que apenas os blocos efetivamente modificados precisam ser armazenados novamente, reduzindo o consumo de espaço e possibilitando a recuperação de estados anteriores do sistema (XIAO et al., 2009).

No mecanismo *Redirect-On-Write*, RoW, quando ocorre uma modificação em um bloco de dados que já está sendo referenciado por um *snapshot*, o sistema não realiza a cópia prévia do bloco original. Em vez disso, o bloco existente permanece inalterado em sua posição no disco, preservando a versão anterior dos dados. A nova escrita é então direcionada diretamente para um novo bloco, localizado em outra região do armazenamento. Dessa forma, o *snapshot* continua apontando para o bloco original, enquanto a versão atualizada do arquivo passa a referenciar o novo bloco que contém as modificações (XIAO et al., 2009).

Essa abordagem reduz a sobrecarga inicial presente no modelo CoW, pois elimina a necessidade de ler e copiar o bloco original antes de registrar a nova versão. Como resultado, o RoW tende a apresentar melhor desempenho em cenários caracterizados por operações intensivas de escrita, nos quais a penalidade da primeira modificação poderia impactar significativamente a performance do sistema (XIAO et al., 2009).

Por outro lado, o uso do RoW pode introduzir maior complexidade nas operações de leitura. Como os dados modificados são gravados em novos blocos distribuídos pelo disco, a estrutura lógica de um arquivo pode tornar-se mais fragmentada ao longo do tempo. Dessa forma, para reconstruir o estado atual de um arquivo, o sistema pode precisar acessar blocos localizados em diferentes regiões do armazenamento, o que pode aumentar o custo das operações de leitura e gerenciamento de dados (XIAO et al., 2009).

Nesse sentido, enquanto o CoW prioriza a preservação das versões anteriores por meio da cópia explícita dos blocos modificados, o RoW busca otimizar o desempenho das operações de escrita ao redirecionar diretamente as modificações para novos blocos, mantendo intactos os dados previamente armazenados (XIAO et al., 2009).

3.3 Sistema de arquivos modernos usando CoW

Os sistemas de arquivos modernos, como Btrfs e *ZFS*, adotam o modelo *Copy-on-Write* para organização e persistência de dados, utilizando a escrita em novos blocos e a atualização de ponteiros como estratégia para manter a consistência do sistema. Essa abordagem permite maior confiabilidade diante de falhas e viabiliza funcionalidades como *snapshots* e versionamento eficiente, ao mesmo tempo em que introduz implicações estruturais e de desempenho que serão discutidas a seguir a partir da análise desses sistemas (BENITES; FLORES; RODRIGUES, 2025).

3.3.1 *B-tree File System* (Btrfs)

O *B-tree File System* (Btrfs) é um sistema de arquivos moderno desenvolvido para o ecossistema Linux com o objetivo de oferecer maior escalabilidade, integridade de dados e recursos avançados de gerenciamento de armazenamento. Sua arquitetura foi projetada para operar de forma integrada com o modelo CoW, permitindo que modificações em dados e metadados sejam registradas em novos blocos, preservando as versões anteriores e garantindo maior consistência em cenários de falha (BENITES; FLORES; RODRIGUES, 2025).

Internamente, a Btrfs organiza o estado persistente do sistema por meio de múltiplas árvores-B, responsáveis por estruturar diferentes tipos de informações, como arquivos, metadados, extensões e verificações de integridade. Essa estrutura utiliza uma variação otimizada para ambientes CoW, permitindo que alterações em nós da árvore sejam registradas sem a necessidade de reconstrução completa da estrutura. Além disso, o sistema emprega o conceito de *extents*, que representam sequências contíguas de blocos em disco, favorecendo operações de leitura e escrita sequenciais e contribuindo para um uso mais eficiente do espaço de armazenamento (BENITES; FLORES; RODRIGUES, 2025).

Uma das características mais relevantes do Btrfs é o suporte nativo a *snapshots* e clones, possibilitado pela própria natureza do CoW. Nesse mecanismo, a criação de um *snapshot* consiste essencialmente em registrar uma nova referência para a raiz da árvore de dados em um determinado momento, permitindo que diferentes versões compartilhem blocos físicos inalterados e armazenem apenas as modificações subsequentes. Essa estratégia reduz significativamente o consumo de espaço e viabiliza operações rápidas de versionamento e recuperação de estados anteriores do sistema (CHEN et al., 2014).

Outro aspecto importante é o mecanismo de verificação de integridade, baseado no uso de *checksums* para dados e metadados. Esses valores de verificação permitem identificar corrupções silenciosas durante operações de leitura e, em configurações que utilizam múltiplos dispositivos de armazenamento, possibilitam a recuperação automática dos dados corretos a partir de cópias redundantes (CHEN et al., 2014).

Apesar das vantagens oferecidas por essa arquitetura, o uso do modelo CoW também introduz desafios relacionados ao desempenho. Como o sistema é estruturado em árvores de blocos, a modificação de um único dado pode exigir a atualização de múltiplos níveis da estrutura, desde o nó folha até a raiz da árvore. Esse processo pode gerar amplificação de escrita, na qual a quantidade de dados efetivamente gravados no disco se torna significativamente maior do que o volume originalmente modificado pela aplicação. Ainda assim, o Btrfs permanece como uma solução relevante em ambientes que demandam mecanismos robustos de integridade, versionamento e gerenciamento avançado de armazenamento (CHEN et al., 2014).

3.4 Arquivos binários

No contexto do desenvolvimento de software e, especialmente, na indústria de jogos digitais, arquivos binários representam grande parte do volume de dados de um projeto. Esse

conjunto inclui ativos como texturas, modelos tridimensionais, áudios, vídeos e outros recursos utilizados pela aplicação. Diferentemente dos arquivos de texto, os binários são compostos por sequências de *bytes* que não possuem significado direto para o sistema operacional, sendo interpretados apenas pela aplicação que os manipula (HOUSTON, 2019).

Sistemas tradicionais de controle de versão, como o Git, operam de forma eficiente com arquivos de texto, pois conseguem identificar modificações por meio de comparações linha a linha. No entanto, esse mecanismo não funciona adequadamente para arquivos binários. Pequenas alterações em um binário podem resultar em mudanças significativas na sua representação em bytes, impedindo que ferramentas de comparação identifiquem diferenças de forma granular. Como consequência, o sistema tende a armazenar novas cópias completas do arquivo a cada modificação, o que pode causar crescimento acelerado do repositório (CHEN et al., 2025).

3.4.1 Desafios no versionamento de arquivos binários

Uma alternativa para lidar com esse problema consiste em utilizar mecanismos de armazenamento baseados em blocos, mencionado anteriormente. Nessa abordagem, quando um arquivo é modificado, o sistema grava os dados alterados em novos blocos de armazenamento, preservando os blocos originais para manter versões anteriores. Apesar das vantagens desse modelo, sua aplicação a arquivos binários apresenta algumas limitações (KASAMPALIS, 2010).

Uma das principais dificuldades está relacionada à propagação de alterações dentro do arquivo. Em formatos binários cuja estrutura não é estática, a inserção ou remoção de dados pode deslocar todos os bytes subsequentes no arquivo. Para o sistema de arquivos, isso faz com que múltiplos blocos pareçam ter sido modificados, mesmo que a alteração real tenha ocorrido apenas em uma pequena parte do conteúdo. Nesses casos, o mecanismo CoW pode ser obrigado a registrar novamente grande parte do arquivo, reduzindo a eficiência do versionamento diferencial (KASAMPALIS, 2010).

Outro fator relevante está relacionado à forma como os sistemas de armazenamento organizam os dados em blocos de tamanho fixo, geralmente na ordem de alguns kilobytes. Escritas que não estão alinhadas com esses blocos podem gerar fragmentação interna ou exigir operações adicionais de leitura e escrita para reconstruir o conteúdo completo do bloco modificado (KASAMPALIS, 2010).

3.4.2 Versionamento baseado em blocos

Embora qualquer arquivo binário possa ser armazenado em blocos, a eficiência do versionamento depende fortemente da estrutura interna do formato utilizado. Segundo Zheng et al. (2016) arquivos que possuem layout previsível e segmentado tendem a se adaptar melhor a estratégias baseadas em CoW ou RoW. Nesses casos, diferentes partes do arquivo são organizadas em regiões bem definidas, permitindo que alterações localizadas não afetem o restante do conteúdo.

Outro aspecto importante é o uso de referenciamento por deslocamento, *offsets*, dentro do arquivo. Formatos que utilizam ponteiros internos para localizar dados permitem que segmentos específicos do binário sejam modificados sem a necessidade de reorganizar toda a estrutura do arquivo. Isso favorece o compartilhamento de blocos não modificados entre diferentes versões.

3.4.3 Limitações do versionamento por blocos em arquivos binários

Por outro lado, alguns tipos de arquivos binários apresentam características que tornam o versionamento por blocos pouco eficiente. Um exemplo comum são arquivos que utilizam compressão global, como formatos compactados ou certas imagens e mídias. Nesses casos, pequenas alterações no conteúdo original podem gerar uma sequência de bytes completamente diferente após a recompressão, fazendo com que o sistema de arquivos interprete todos os blocos como modificados (CHEN et al., 2025).

Situação semelhante ocorre com arquivos que utilizam criptografia, nos quais alterações mínimas no dado original produzem resultados significativamente diferentes no conteúdo cifrado. Além disso, formatos que possuem forte dependência estrutural entre seus dados também dificultam o reaproveitamento de blocos entre versões (ZHENG et al. 2016).

Dessa forma, a viabilidade do versionamento eficiente de arquivos binários depende diretamente das características estruturais do formato utilizado. Arquivos que preservam a organização interna dos dados e permitem modificações localizadas tendem a aproveitar melhor os mecanismos de armazenamento baseados em blocos, enquanto formatos altamente compactados ou encadeados reduzem significativamente os benefícios dessas abordagens (CHEN et al., 2025).

3.5 O formato glTF

O glTF (*GL Transmission Format*) consiste em um formato de arquivo aberto e livre de *royalties*, desenvolvido pelo Khronos Group, consórcio industrial sediado em Beaverton e responsável pela criação de padrões como WebGL e Vulkan. O formato é projetado para a transmissão e o carregamento eficiente de cenas e modelos tridimensionais por aplicações, sendo reconhecido como um padrão de entrega final para ativos em três dimensões. Essa classificação vem de sua arquitetura voltada à minimização tanto do tamanho dos arquivos quanto do esforço computacional necessário em tempo de execução para o processamento e a renderização dos dados de geometria, textura e animação (HOUSTON, 2019).

O desenvolvimento do glTF teve início em março de 2012, quando o projeto foi proposto por membros do grupo COLLADA com o objetivo de criar um formato que permitisse a entrega padronizada de conteúdo 3D para aplicações WebGL. A primeira especificação do padrão foi ratificada em outubro de 2015. Posteriormente, o lançamento da versão 2.0 representou um marco para a indústria ao introduzir o suporte nativo a materiais baseados em física, *Physically Based Rendering* (PBR), garantindo a consistência visual dos ativos entre diferentes motores de renderização (HOUSTON, 2019; KHRONOS GROUP, 2022).

3.5.1 Estrutura geral de um ativo glTF

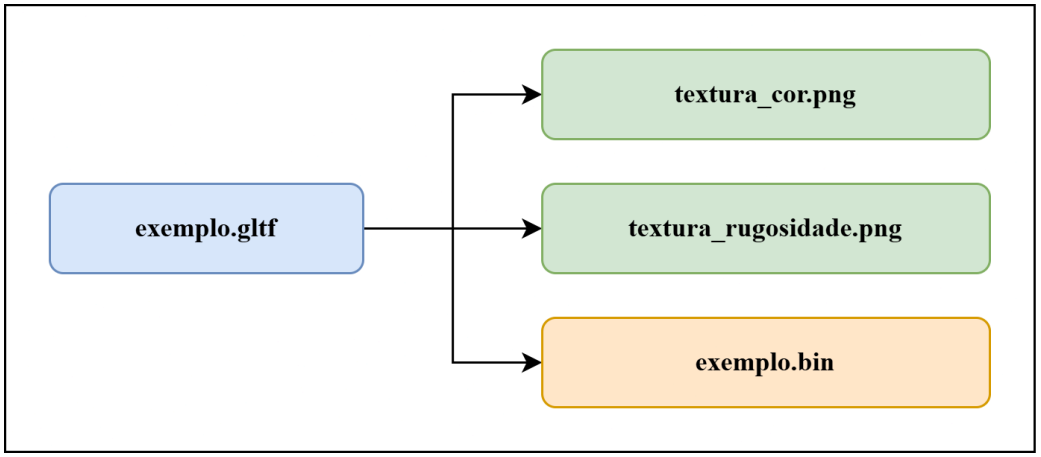
Um ativo glTF não é necessariamente um arquivo único, mas um ecossistema modular composto por três pilares fundamentais que se relacionam para descrever uma cena ou objeto tridimensional. Segundo Khronos Group (2022), podemos dividir esse formato nos seguintes componentes:

a) O Arquivo de Descrição (.gltf) consiste em um arquivo estruturado em formato JSON responsável por representar a estrutura lógica da cena. Nele são definidos elementos como a hierarquia de nós, materiais, câmeras, animações e referências aos demais recursos utilizados no modelo.
b) O arquivo binário (.bin) atua como contêiner de dados para armazenar informações estruturais, como geometria de vértices e índices, quadros chave de animação e informações de *skinning* que permitem a deformação de malhas em modelos animados.

c) Os Arquivos de Imagem (.jpg, .png) são responsáveis por armazenar as texturas utilizadas para definir a aparência superficial dos modelos, incluindo características visuais como cor, rugosidade e propriedades de iluminação.

A Figura 3 ilustra visualmente a arquitetura modular do formato glTF. Conforme detalhado, o arquivo de descrição atua como o orquestrador central, definindo a estrutura da cena e referenciando os demais componentes. Ele se conecta ao arquivo binário, que armazena os dados geométricos e de animação essenciais, e aos arquivos de imagem, que fornecem as texturas para a renderização visual.

Figura 3 - A estrutura básica de um glTF.



Fonte: Autoria própria (2026).

3.5.2 O contêiner GLB

Alternativamente, o formato suporta o contêiner .glb, que consolida todos esses componentes em um único arquivo binário e utiliza uma estrutura de *chunks* para facilitar o transporte e evitar múltiplas requisições de rede (KHRONOS GROUP, 2021).

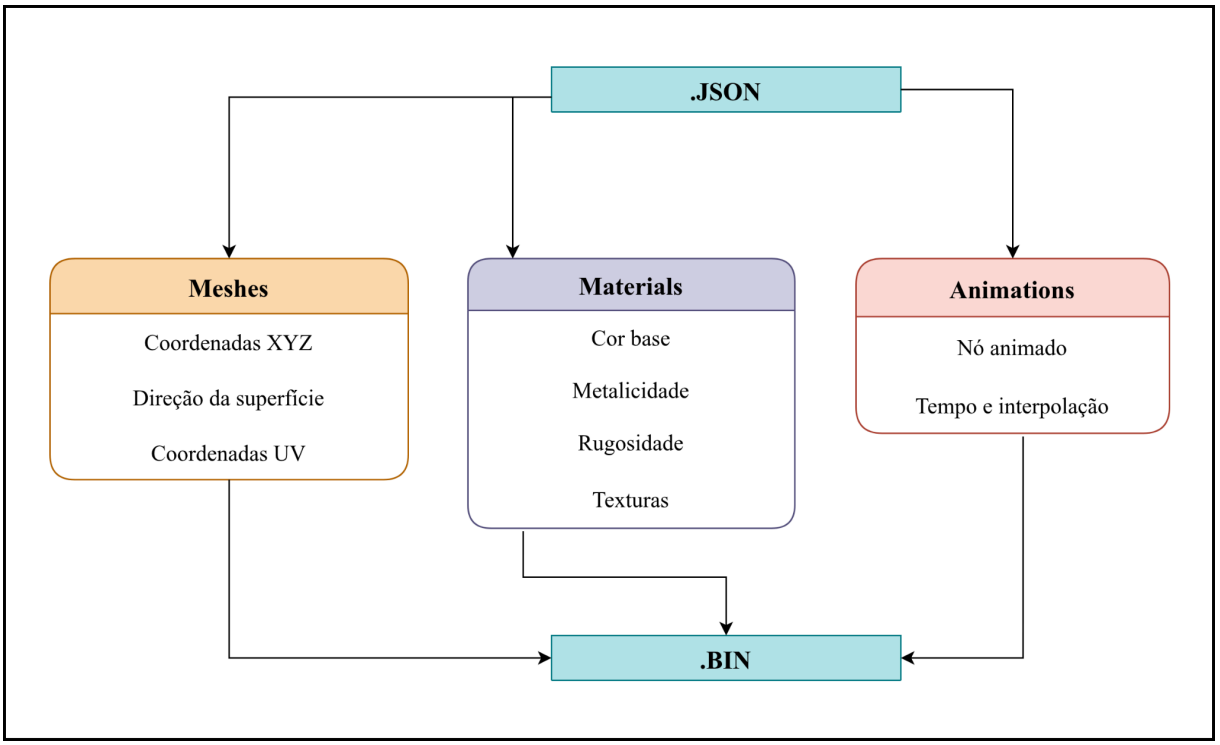
O arquivo JSON é o esqueleto que define a estrutura do binário impondo restrições rígidas para simplificar a implementação. Com isso, deve usar codificação UTF-8 sem conter a sequência de bytes inicial; as chaves de objetos devem ser únicas e nomes de propriedades devem usar apenas caracteres ASCII. A organização interna do JSON é feita através de arrays de objetos, onde a comunicação entre os diferentes elementos ocorre obrigatoriamente através de índices. Nesse sentido, pode-se entender como referência à posição do objeto no *array*. Os principais elementos descritos são *scenes* e *nodes* e eles definem a estrutura básica e a hierarquia da cena (KHRONOS GROUP, 2021).

Além dos elementos estruturais de nós e cenas, o arquivo JSON do glTF também define componentes que modulam a cena em sua totalidade, detalhando aspectos visuais e comportamentais dos objetos tridimensionais (Khronos Group, 2021):

a) As *meshes* descrevem a geometria fundamental dos objetos através de primitivas de renderização, que definem se o ativo deve ser desenhado como pontos, linhas ou triângulos. Internamente, essas primitivas referenciam atributos de vértices, como a posição e os vetores normais, utilizando índices de *accessors* que apontam para os dados brutos no arquivo binário.
b) Os *materials* especificam as propriedades ópticas dos modelos, incluindo mapeamentos de cor base, metalicidade, rugosidade e texturas de normais. Esses materiais definem como a luz reage à superfície e à profundidade da malha, influenciando diretamente a aparência visual do objeto.
c) *Animations* operam através de uma estrutura de canais e samplers. Os canais definem o alvo da animação, como a translação, rotação ou escala de um nó, ou os pesos de um ponto de deformação. Os *samplers*, por sua vez, gerenciam os dados de tempo e os valores correspondentes.
d) As *cameras* estabelecem as configurações de projeção necessárias para a visualização da cena pela aplicação.

   Podemos entender melhor as divisões do arquivo JSON na Figura 4 abaixo.

Figura 4 - A estrutura básica de um glTF.



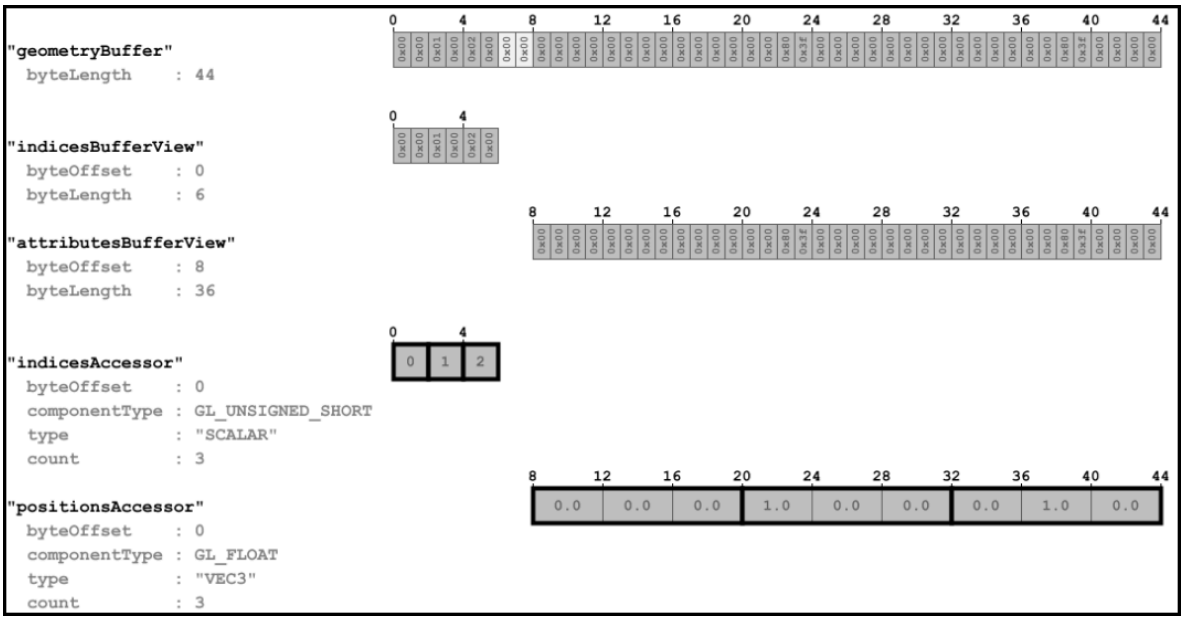
Fonte: Autoria própria (2026).

Uma funcionalidade crítica do JSON é o referenciamento de recursos externos por meio de *Uniform Resource Identifiers* (URIs), o que permite que as texturas e os dados binários residam em arquivos separados. Para acessar e interpretar os dados brutos contidos no arquivo binário, o glTF utiliza uma estrutura de três camadas interdependentes dentro do JSON, garantindo a organização e a leitura eficiente das informações (KHRONOS GROUP, 2022).

A primeira camada é composta pelos *buffers*, que apontam para o recurso binário completo e definem o seu tamanho total em *bytes*. Em seguida, as *bufferViews* segmentam esse buffer em fatias menores, identificando quais partes pertencem especificamente aos dados de vértices, índices ou animações. Por fim, os *accessors* fornecem a interpretação tipada dos dados dentro de uma *bufferView*, definindo, por exemplo, que uma determinada sequência de *bytes* deve ser lida como um vetor de três números flutuantes. Essa hierarquia permite que a aplicação decodifique a geometria e os comportamentos do modelo de forma estruturada (KHRONOS GROUP, 2022). A aplicação prática dessa hierarquia pode ser observada na Figura 5, que ilustra a decomposição de um triângulo no plano XY através das três camadas mencionadas. Na base da estrutura, o *geometryBuffer* representa o arquivo binário completo em uma sequência de 44 *bytes* brutos. Essa unidade é segmentada pelas *bufferViews*, que recortam regiões específicas para os índices dos triângulos e para os

atributos de vértice. Por fim, os *accessors* realizam a interpretação tipada desses dados, convertendo os *bytes* em valores compreensíveis para a aplicação, como vetores de três números flutuantes que definem as coordenadas espaciais dos vértices na origem e nos eixos correspondentes. Esse processo demonstra como o formato glTF organiza a geometria de forma otimizada, permitindo que dados hexadecimais complexos sejam transformados em elementos visuais tridimensionais com baixo custo de processamento.

Figura 5 - Representação da hierarquia de camadas em um modelo.



Fonte: Khronos Group (2026).

3.5.3 Arquivos similares

Assim como o glTF utiliza *buffers* e *bufferViews* para organizar dados brutos em pedaços lógicos independentes, outros formatos de arquivo possuem abordagens semelhantes. O *Tag Image File Format* (TIFF), por exemplo, conforme especificado pela Aldus Corporation (1992), é um formato de imagem rasterizada estruturado em etiquetas, que descrevem e armazenam dados de imagem através de campos lógicos com identificadores e valores. Sua criação garante independência e portabilidade, não estando vinculado a processadores, sistemas operacionais ou *hardware* específicos, o que o torna possivelmente extensível ao nosso método. A organização modular do TIFF é evidenciada pelo seu cabeçalho, que aponta para um diretório de arquivos de imagem. Este diretório contém informações cruciais sobre a imagem e, mais importante, os ponteiros, *offsets* como vimos no

glTF, para a localização real dos dados brutos no arquivo. O formato suporta a segmentação da imagem em pedaços menores, permitindo que esses segmentos sejam armazenados de forma fragmentada e acessados aleatoriamente e eficientemente (Aldus Corporation, 1992).

De forma análoga, o formato WAVE (*Waveform Audio File Format*) para áudio, conforme especificações conjuntas da IBM Corporation e Microsoft Corporation (1991), organiza seus dados brutos em segmentos lógicos independentes. O WAVE adota uma arquitetura etiquetada que promove um arquivo compatível. O arquivo é composto por blocos, cada um com um identificador único e um campo indicando seu tamanho, permitindo que aplicações ignorem blocos não reconhecidos. Para ser válido, um arquivo WAVE deve conter obrigatoriamente um bloco de formato, que define as propriedades do áudio, e um bloco de dados, com as amostras binárias brutas. O bloco de formato detalha parâmetros técnicos essenciais, como categoria do formato, número de canais, taxa de amostragem e alinhamento de blocos.

3.6 Trabalhos relacionados

Esta seção apresenta uma análise de trabalhos relacionados que abordam o controle de versão e histórico de edição em contextos de imagens e modelos 3D.

3.6.1 Controle de revisão não linear para imagens

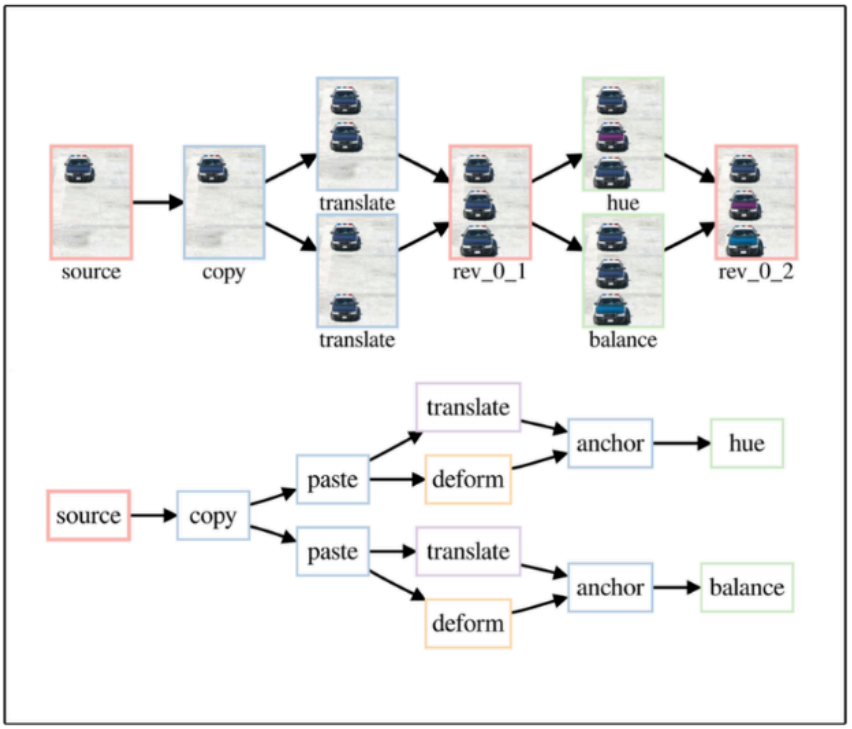
O trabalho desenvolvido por Chen, Wei e Chang (2011) apresenta um sistema de controle de versão para imagens digitais, que se diferencia dos métodos tradicionais ao registrar as ações de edição do usuário em vez de armazenar o arquivo completo a cada versão. Este enfoque visa otimizar o armazenamento e introduzir funcionalidades avançadas de manipulação do histórico de edições.

O funcionamento do sistema baseia-se em duas estruturas primárias, o Grafo Acíclico Dirigido (DAG) e o Grafo de Revisão (RevG). O DAG constitui o cerne do sistema, onde cada nó representa uma operação de edição, como pinceladas ou ajustes de cor, e as arestas estabelecem as dependências temporais, espaciais e semânticas entre essas ações. Ações que afetam diferentes partes da imagem são dispostas em caminhos paralelos, permitindo um histórico de edição não linear. O RevG, por sua vez, oferece uma visualização de múltiplas resoluções do DAG, utilizando miniaturas para representar o progresso das edições e facilitar a navegação no histórico, a criação de ramificações e a fusão de versões. Para o registro e

reprodução das edições, o sistema emprega um sistema de *log*, que captura discretamente as ações do usuário em softwares como o GIMP, e um método capaz de reconstruir qualquer estado da imagem a partir desses registros. Um filtro de importância visual também é utilizado para agrupar nós do DAG no RevG, priorizando a relevância das ações para uma interface mais limpa (CHEN; WEI; CHANG, 2011).

A Figura 6, demonstra o uso da estrutura de grafos para armazenar as edições na imagem. A partir de uma imagem original, são realizadas operações de edição, como cópia, translação, deformação e ajustes de cor, gerando diferentes revisões da imagem. Essas ações são organizadas na estrutura DAG, que representa as dependências temporais, espaciais e semânticas entre as edições, permitindo visualizar e gerenciar o histórico de modificações de forma não linear (CHEN; WEI; CHANG, 2011).

Figura 6 - Alterações em uma imagem de entrada.



Fonte: Chen, Wei e Chang (2011).

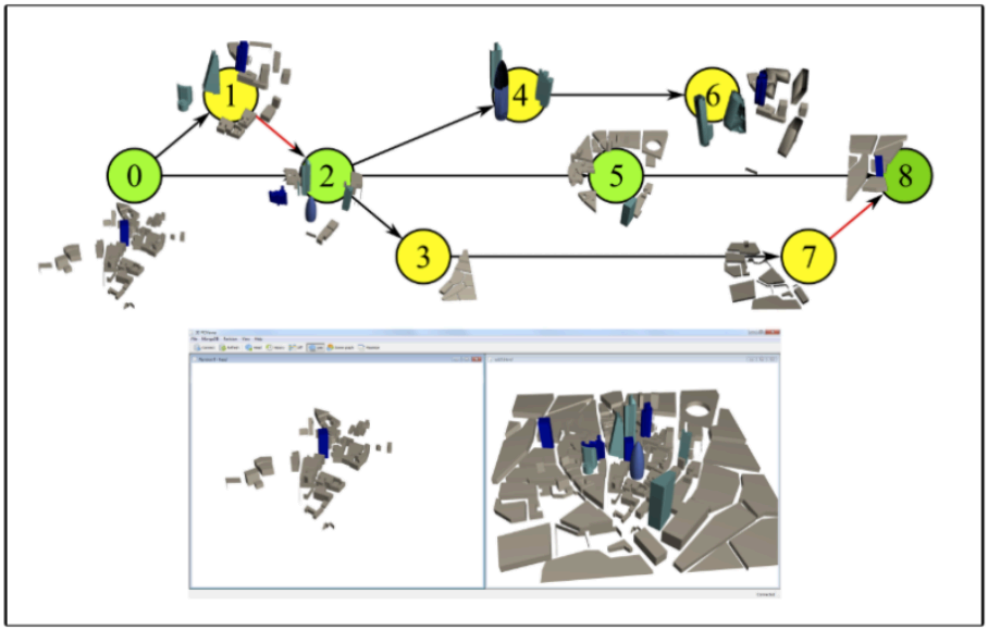
3.6.2 *Framework* de controle de revisão 3D

Doboš e Steed (2012) propõem um sistema de controle de versão para ativos 3D, com foco em cenas massivas e edição colaborativa. Este sistema utiliza um banco de dados NoSQL, MongoDB, como repositório central. A cena 3D é desconstruída em um Grafo de Cena, onde componentes como malhas, materiais e câmeras são armazenados como

documentos binários individuais no formato BSON. O versionamento ocorre no nível do documento, garantindo que apenas as partes alteradas da cena sejam salvas em novas revisões, otimizando o uso de espaço.

O *framework* opera através de duas interfaces principais: uma para o usuário, que gerencia revisões e resolução de conflitos, e um cliente web somente leitura que utiliza WebGL para visualização remota. O método de gerenciamento de revisões é estruturado por dois grafos, sendo o primeiro o Grafo de Cena, que representa a estrutura lógica do modelo 3D, e o Histórico de Revisão, que acompanha a evolução temporal do projeto, suportando ramificações e fusões. A estrutura do histórico de revisões, ilustrada na Figura 7, organiza o versionamento como um grafo acíclico dirigido, onde cada nó representa uma revisão com suas respectivas mudanças incrementais sobre o grafo de cena. A linha principal de desenvolvimento evolui linearmente, enquanto ramificações permitem linhas de desenvolvimento paralelas, que podem ser reintegradas através de operações de *merge*. Este modelo de versionamento, que armazena apenas as mudanças incrementais dos nós do grafo de cena, reduz significativamente o custo de armazenamento em comparação com sistemas tradicionais que tratam arquivos binários 3D como unidades atômicas. Para recuperar uma versão específica, o algoritmo percorre o histórico de ancestrais, selecionando a versão mais recente de cada objeto dentro da linhagem. No processo de fusão, uma ferramenta de comparação entre objetos tridimensionais é empregada para identificar e visualizar conflitos.

Figura 7 - Estrutura do Histórico de Revisões e Evolução do Modelo.



Fonte: Doboš e Steed (2012).

3.6.3 Suporte a compressão delta em sistemas de arquivos

MacDonald (2000) propõe o *Xdelta File System* (XDFS), um sistema de arquivos desenvolvido para gerenciar de forma eficiente o armazenamento e o transporte de versões de arquivos binários por meio de compressão delta, técnica que registra apenas as diferenças entre duas versões de um arquivo em vez de armazenar cópias completas.

O sistema utiliza o algoritmo *Xdelta*, que emprega funções de impressão digital sobre sequências de *bytes* para identificar trechos correspondentes entre versões e gerar um pacote contendo apenas as modificações. Para garantir que o tempo de inserção de uma nova versão seja independente do tamanho total do histórico armazenado, o XDFS adota um mecanismo de transações baseado em banco de dados, resolvendo um problema presente em sistemas anteriores, nos quais o arquivo de histórico precisava ser reescrito integralmente a cada atualização.

O trabalho oferece duas estratégias complementares de armazenamento. A primeira prioriza a velocidade de recuperação, garantindo que qualquer versão possa ser reconstruída com a aplicação de no máximo um pacote de diferenças. A segunda prioriza a economia de espaço, encadeando as diferenças em ordem inversa a partir da versão mais recente. O sistema também foi projetado para eficiência na transmissão de dados em rede, permitindo extrair diferenças entre quaisquer duas versões sem reconstruí-las integralmente (MACDONALD, 2000).

Apesar das vantagens em termos de desempenho no processamento de arquivos binários, o XDFS apresenta limitações relevantes. O custo de metadados gerado pelo banco de dados interno pode superar o tamanho dos próprios dados comprimidos, e o processo de criação de um novo histórico é significativamente mais lento do que em sistemas de arquivos convencionais. Além disso, assim como discutido na seção 3.5.3 deste trabalho, o sistema não lida adequadamente com arquivos que possuem compressão interna, pois pequenas alterações no conteúdo original resultam em mudanças massivas nos *bytes* armazenados, anulando os benefícios da abordagem diferencial (MACDONALD, 2000).

3.6.4 Análise comparativa dos trabalhos relacionados

Os trabalhos apresentados nas seções anteriores abordam o versionamento de arquivos sob perspectivas distintas, e esta seção traça um paralelo entre essas abordagens e a proposta deste trabalho.

Doboš e Steed (2012) adotam um banco de dados NoSQL como repositório central, exigindo a substituição do sistema de arquivos por uma infraestrutura dedicada. O presente trabalho segue o caminho oposto, aproveitando as capacidades nativas de sistemas de arquivos modernos para realizar o armazenamento diferencial de forma transparente, sem dependências externas. Além disso, enquanto eles segmentam o modelo segundo sua estrutura lógica de cena, a abordagem aqui proposta utiliza os segmentos internos definidos pelo próprio formato do arquivo como unidade de versionamento.

MacDonald (2000) propõe uma solução baseada em compressão delta que trata qualquer binário como uma sequência opaca de bytes, sendo agnóstica ao formato do arquivo. A proposta deste trabalho difere em dois aspectos, em vez de calcular diferenças byte a byte, utiliza funções de resumo criptográfico para identificar segmentos inalterados, delegando ao sistema de arquivos o compartilhamento físico desses blocos. E ao segmentar o binário segundo sua estrutura interna, garante que uma alteração localizada não produza mudanças em cascata por todo o conteúdo.

Chen, Wei e Chang (2011) versiona não os dados do arquivo, mas o registro das ações de edição do usuário, o que exige integração direta com o software de edição. O presente trabalho atua em uma camada distinta, tendo como objeto de versionamento os próprios bytes resultantes da edição, tornando a abordagem compatível com qualquer fluxo de trabalho de produção sem instrumentação adicional.

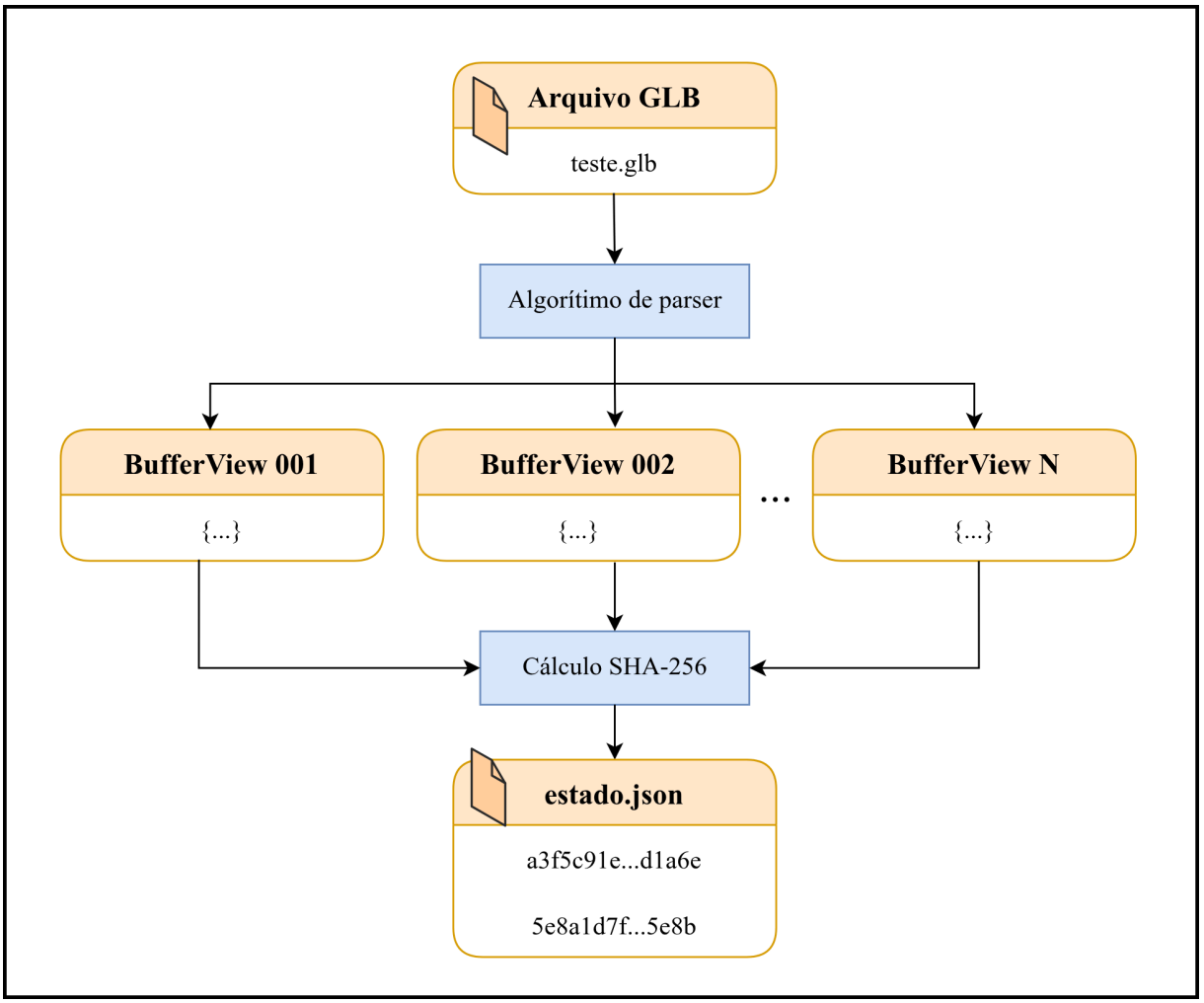
Em síntese, o que diferencia a proposta deste trabalho dos demais é a combinação entre sensibilidade à estrutura interna do arquivo e o uso de mecanismos nativos do sistema de arquivos, partindo do princípio de que arquivos binários com segmentação interna bem definida são candidatos naturais ao versionamento eficiente por blocos.

4 METODOLOGIA

A metodologia utilizada neste projeto é a experimentação. Com base nisso, a abordagem proposta baseia-se no desenvolvimento de um mecanismo de versionamento diferencial otimizado para arquivos GLB, fundamentado na fragmentação do binário em blocos lógicos. Utilizando a estrutura de *bufferViews* do glTF como critério de segmentação, a abordagem permite identificar e armazenar apenas as seções que sofreram alterações, como geometria ou animações, em vez de replicar o arquivo integralmente. Essa estratégia aproveita a arquitetura modular do formato para reduzir a redundância nos repositórios, visando validar a eficiência do versionamento por blocos em termos de economia de armazenamento e

desempenho sistêmico. A Figura 8 ilustra o fluxo geral de como o GLB é tratado no método, e será explicada mais detalhadamente ao longo da seção de metodologia.

Figura 8 - Fluxo de observação do arquivo.



Fonte: Autoria própria (2026).

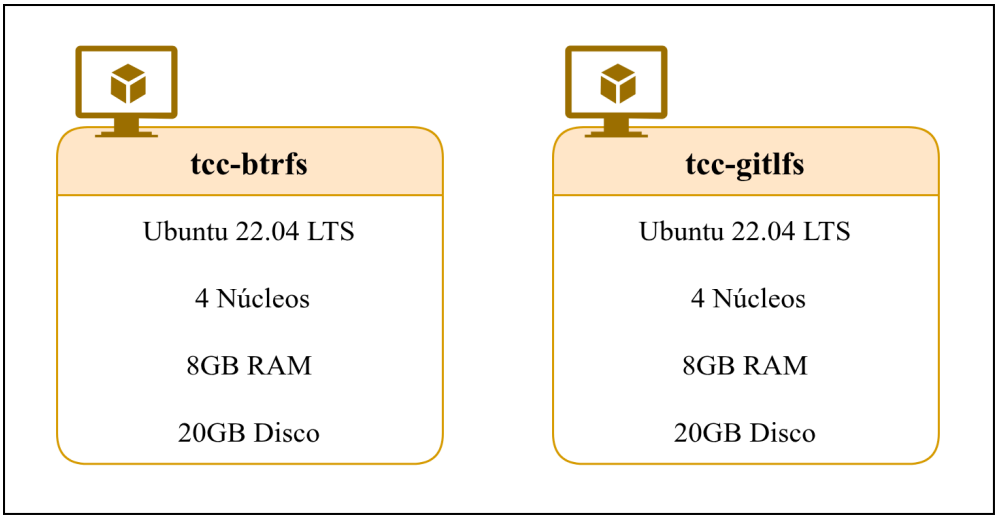
4.1 Ambiente de experimentação

Para a execução dos testes e a validação do mecanismo de versionamento diferencial, serão configurados dois ambientes virtualizados e isolados utilizando o *Multipass*, ferramenta de gerenciamento de máquinas virtuais desenvolvida pela Canonical para gerenciamento de máquinas virtuais Ubuntu (CANONICAL, 2026). A separação em duas instâncias distintas foi adotada para garantir que as métricas coletadas reflitam exclusivamente o comportamento de cada método, sem interferências externas entre os processos de versionamento avaliados.

Na Figura 9 é possível visualizar o ambiente de teste proposto em duas máquinas virtuais. A primeira instância, denominada *tcc-btrfs*, será destinada à execução da abordagem

proposta, utilizando o sistema de arquivos Btrfs como camada de armazenamento diferencial com suporte nativo a *snapshots* baseados no mecanismo CoW. A segunda instância, denominada *tcc-gitlfs*, será dedicada à execução do método de referência, baseado no Git LFS. Ambas as máquinas virtuais serão criadas com o sistema operacional Ubuntu 22.04 LTS, garantindo a estabilidade e a reprodutibilidade dos experimentos, e receberão a mesma configuração de hardware, 4 núcleos de processamento, 8 *gigabytes* de memória RAM e 20 *gigabytes* de armazenamento em disco.

Figura 9 - Configuração das máquinas virtuais para experimentação.



Fonte: Autoria própria (2026).

Para a implementação do algoritmo de segmentação e análise dos arquivos GLB, será utilizada a linguagem *Python* com os módulos nativos *struct* e *json*, responsáveis respectivamente pela leitura dos binários do contêiner GLB e pela interpretação da estrutura JSON que descreve as *bufferViews* do modelo. Complementarmente, o Git LFS será configurado na segunda instância para servir como método de referência, permitindo medir a eficácia da abordagem proposta em relação a um padrão de versionamento consolidado na indústria. Os modelos tridimensionais utilizados nos testes foram obtidos a partir do repositório oficial de amostras do Khronos Group (KHRONOS GROUP, 2026).

4.2 Procedimentos Experimentais

Os procedimentos experimentais serão organizados de forma a permitir uma comparação direta e controlada entre a abordagem proposta e o método de referência. Para isso, ambos os ambientes receberam o mesmo modelo tridimensional como entrada e serão

submetidos ao mesmo conjunto de versões modificadas, geradas com parâmetros idênticos e reproduzíveis. A seguir são descritos o objeto de teste selecionado, o processo de geração controlada das versões, o fluxo de execução de cada método e as métricas utilizadas para avaliação dos resultados.

4.2.1 Objeto de teste

A escolha do modelo tridimensional *ABeautifulGame.glb* como objeto de teste para este estudo fundamenta-se nas características que o tornam particularmente adequado para o versionamento por blocos, conforme delineado na seção 3.4.1 do referencial teórico. Este modelo atende aos critérios de um layout previsível e de uma segmentação interna bem definida por *bufferViews*, o que é crucial para a aplicação eficiente de estratégias de armazenamento diferencial. Adicionalmente, o modelo empregado não tem compressão global nem criptografia, fatores que, segundo a literatura, poderiam comprometer a granularidade e a eficácia do versionamento baseado em blocos. O modelo escolhido pode ser observado na Figura 10.

Figura 10 - Cena tridimensional disponibilizada em Khronos Group.

Fonte: Khronos Group (2026).

É importante ressaltar uma limitação inerente ao modelo selecionado que é a ausência de animações. Consequentemente, os cenários de teste serão concentrados exclusivamente em modificações de geometria, permitindo uma análise aprofundada do comportamento do sistema de versionamento diante de alterações estruturais nos dados tridimensionais. Quantitativamente, o modelo apresenta um tamanho de 42,9 MB e é composto por 108

bufferViews, das quais 75 são dedicadas à geometria e as 33 restantes a índices e dados estruturais.

4.2.2 Geração controlada das versões

Para a criação das versões modificadas do objeto de teste, um *script* de modificação será desenvolvido especificamente para simular edições típicas realizadas por um artista 3D sobre a geometria de um modelo. As alterações serão aplicadas diretamente nos valores de ponto flutuante de 32 bits contidos nas *bufferViews* de geometria do arquivo *ABeautifulGame.glb*. Essa abordagem garante que as modificações mimetizem de forma realista as alterações incrementais que ocorreriam em um fluxo de trabalho de modelagem tridimensional.

A reprodutibilidade dos experimentos será assegurada pela utilização de uma semente fixa para o gerador de números pseudo aleatórios. Este procedimento garante que as sequências de modificações aplicadas sejam idênticas em ambos os ambientes de teste, permitindo uma comparação direta e justa entre a abordagem proposta e o método de referência. A consistência na geração das versões é fundamental para validar a eficácia dos mecanismos de versionamento em cenários controlados.

Foram definidos oito cenários de teste, cada um representando uma intensidade crescente de modificação, variando de 2% a 90% dos dados de geometria. A progressão gradual da intensidade das alterações permite observar o comportamento do método em diversas condições, desde edições pontuais e localizadas até modificações massivas que afetam grande parte da estrutura do modelo. A Tabela 1 detalha os cenários e suas respectivas intensidades de modificação.

Tabela 1 – Cenários de teste e intensidades de modificação.

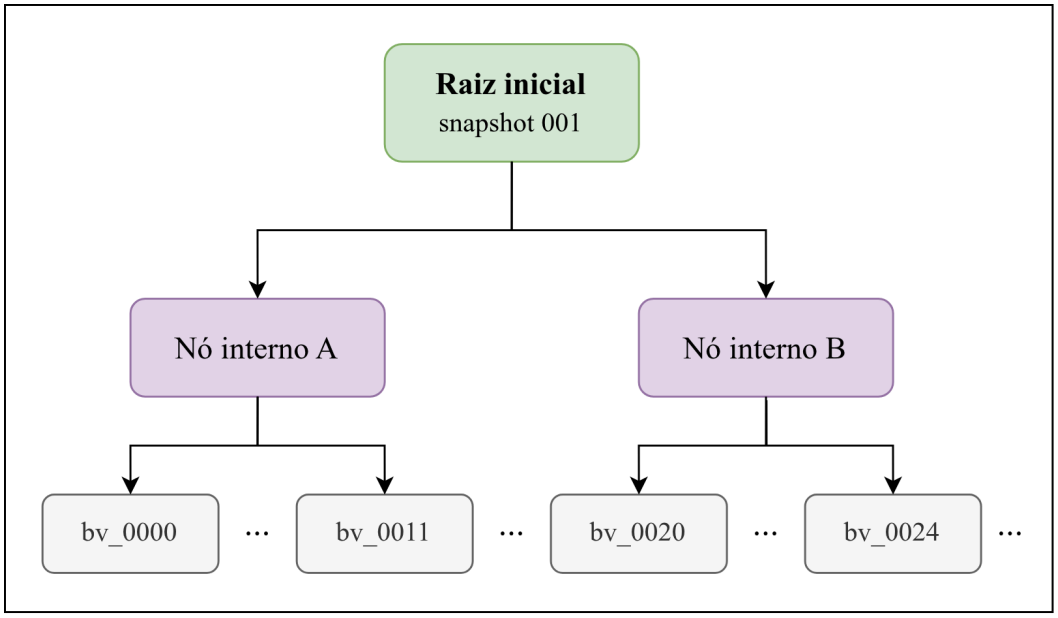
Cenário	Intensidade de Modificação
1	2%
2	5%
3	10%
4	20%
5	30%
6	50%
7	70%
8	90%

Fonte: Autoria própria (2026).

4.2.3 Método proposto

O mecanismo de versionamento desenvolvido, possui duas camadas complementares. Na camada lógica, Figura 8, um módulo de análise implementado em *Python* realiza a leitura e a decomposição do arquivo GLB, enquanto na camada física, que pode ser observada na Figura 11, o sistema de arquivos Btrfs é responsável pelo armazenamento diferencial e pela preservação das versões anteriores por meio de snapshots CoW.

Figura 11 - Camada física de uma versão inalterada.

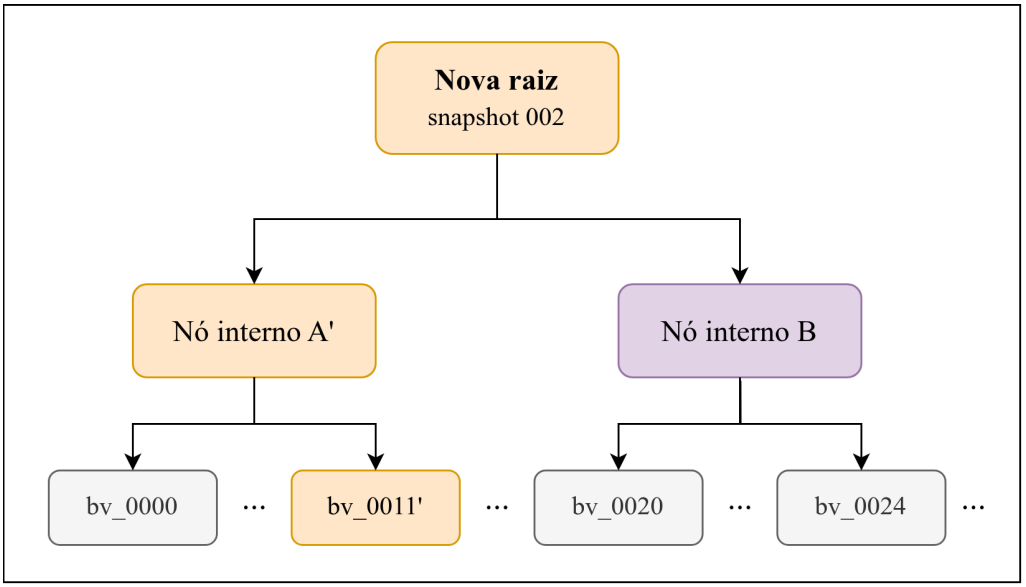


Fonte: Autoria própria (2026).

A Figura 12 ilustra o estado da camada física após o registro de uma nova versão com uma modificação realizada e a Figura 11 mostra a árvore correspondente ao primeiro snapshot, composta pela raiz inicial, pelos nós internos A e B e pelos nós folha contendo as *bufferViews* originais. Na Figura 12, após a detecção de que a *bufferView* **bv_0011** foi modificada, o mecanismo CoW do Btrfs aloca um novo nó folha **bv_0011'** para armazenar o conteúdo atualizado, sem sobrescrever o bloco original. Como a modificação afeta apenas um ramo da árvore, o nó interno A precisa ser copiado para um novo nó A', cujo ponteiro é atualizado para referenciar **bv_0011'** em vez de **bv_0011**. Esse processo de cópia propaga-se até a raiz, resultando em uma nova raiz que representa o estado da segunda snapshot. O nó interno B e todos os nós folha sob seu ramo, incluindo **bv_0020**, **bv_0024** e os demais, permanecem intocados e são referenciados simultaneamente pelos dois snapshots, sem duplicação física no disco. Por consequência, o custo de armazenamento do snapshot **v0002** é

proporcional exclusivamente ao volume de dados que efetivamente se alterou: o novo nó folha, o nó interno copiado e a nova raiz, correspondendo a uma fração mínima do tamanho total do arquivo.

Figura 12 - Camada física de uma versão com alterações.



Fonte: Autoria própria (2026).

Após o entendimento de como as duas camadas do método funcionam se estipula as etapas gerais executadas nas camadas lógica e física. A primeira etapa do método consiste na leitura do contêiner GLB e na separação do mesmo em pedaços. Conforme descrito na seção 3.3.2, o formato GLB organiza seu conteúdo em um JSON, responsável pela descrição da estrutura da cena, e um BIN, que armazena os dados binários brutos. O módulo de análise lê o cabeçalho de 12 *bytes* do arquivo, identifica as porções do arquivo por seus identificadores de tipo e os separa em memória para processamento nos próximos momentos.

A segunda etapa consiste na extração das *bufferViews* como unidades de versionamento. Como apresentado na seção 3.3.1, o JSON do glTF descreve cada *bufferView* por meio de um par de valores, *byteOffset* e *byteLength*, que delimita com precisão a posição e o tamanho de cada segmento dentro do binário. Sendo assim, o módulo extrai cada *bufferView* individualmente, produzindo unidades de dados semanticamente conexas e independentes entre si. No modelo *ABeautifulGame*, utilizado nos experimentos, esse processo resultou em 108 *bufferViews* extraídas de um total de 42.9 *megabytes*, distribuídas em segmentos com tamanhos variando de 9.376 *bytes* a 1.843.200 *bytes*, correspondendo a

atributos de vértices, normais, coordenadas de textura e índices de face de 15 malhas diferentes.

O terceiro passo envolve a criação de uma assinatura digital única para cada *bufferView*. O algoritmo SHA-256 é aplicado sobre os *bytes* de cada segmento, produzindo um identificador de 256 *bits* que representa exclusivamente o conteúdo daquela unidade. Esses *hashes* são armazenados em arquivo JSON ao término de cada operação de versionamento, constituindo o estado de referência utilizado na comparação com a versão subsequente.

A quarta etapa é crucial para o estado lógico do método, consistindo na detecção diferencial entre versões. Ao receber um novo arquivo GLB, o sistema extrai suas *bufferViews*, calcula os *hashes* correspondentes e os compara com os *hashes* da versão anterior. Nesse sentido cada *bufferView* é classificada em uma de quatro categorias, sendo elas, nova, modificada, removida ou inalterada. Apenas as *bufferViews* classificadas como novas ou modificadas são encaminhadas para gravação em disco. As *bufferViews* inalteradas são descartadas do processo de escrita, pois seus dados já estão preservados no subvolume ativo.

A quinta etapa consiste na gravação dos segmentos modificados no subvolume Btrfs e na criação de um *snapshot* imutável. Cada bufferView é armazenada como um arquivo binário independente no subvolume ativo, identificado pelo seu índice no arquivo GLB. Após a gravação de todos os segmentos alterados, o mecanismo nativo de *snapshots* do Btrfs é acionado para registrar o estado atual do subvolume como uma versão imutável de leitura. Conforme discutido nas seções 3.2.1 e 3.3.1, o Btrfs implementa esse mecanismo por meio do modelo CoW, no qual os blocos físicos não modificados entre versões consecutivas são compartilhados entre o subvolume ativo e os *snapshots* anteriores, sem duplicação de dados no disco. Dessa forma, o custo de armazenamento de cada snapshot é proporcional exclusivamente ao volume de dados que efetivamente se alterou, e não ao tamanho total do arquivo versionado.

4.2.4 Conceitos Fundamentais da Camada Física

A camada física do método proposto é fundamentada no sistema de arquivos Btrfs, como pode-se observar na subseção anterior, escolhido por sua capacidade avançada de gerenciamento de dados e versionamento.

Um dos pilares do Btrfs são os subvolumes. Diferente de um diretório comum, um subvolume é uma unidade independente que possui sua própria árvore de dados. Isso significa que ele pode ser tratado como um sistema de arquivos separado, mesmo estando dentro de um volume Btrfs maior. No método acima, utiliza-se subvolumes para organizar o ciclo de versionamento. Um subvolume principal, este chamado versões, atua como um contêiner, e dentro dele, um subvolume ativo, chamado versão atuais, representa a área de trabalho onde os dados das *bufferViews* são gravados e atualizados.

A eficiência do Btrfs no versionamento é alcançada pela combinação entre o modelo CoW e sua organização interna baseada em árvores-B. Diferentemente de sistemas de arquivos tradicionais, que mantêm estruturas de metadados em posições fixas do disco, o Btrfs organiza todo o seu estado persistente, sendo composto por arquivos, diretórios, extensões de dados e *checksums*, em um conjunto de árvores-B aninhadas, todas enraizadas em uma estrutura central denominada árvore raiz. Cada nó interno dessas árvores armazena chaves de busca e ponteiros para os nós filhos, enquanto os nós folha contêm os dados ou metadados propriamente ditos. Essa organização hierárquica permite localizar qualquer item do sistema de arquivos com custo logarítmico em relação ao volume total de dados, tornando as operações de leitura e escrita eficientes mesmo em volumes de grande escala.

A escolha da árvore-B como estrutura fundamental não é aleatória, sendo essencial para que o CoW possa ser implementado de forma eficiente. Quando um bloco de dados é modificado, o Btrfs precisa atualizar não apenas o bloco em si, mas também o caminho de metadados que leva até ele, desde o nó folha até a raiz da árvore. Em uma estrutura de árvore-B, esse caminho é limitado em profundidade, o que significa que a propagação da atualização envolve um número controlado de nós. Cada nó desse caminho recebe uma nova cópia modificada, gravada em uma posição diferente do disco, enquanto as versões anteriores permanecem intactas. Ao término da operação, apenas o ponteiro raiz é atualizado para referenciar a nova versão da árvore. Esse mecanismo, conhecido como *shadow paging*[2] na literatura de sistemas de arquivos, garante que o estado anterior da árvore continue acessível enquanto o novo estado é construído de forma atômica, comportamento que pode ser observado na Figura 12, onde apenas o caminho afetado pela modificação da *bufferView* **bv_0011** recebe novos nós, enquanto os demais permanecem compartilhados entre os dois snapshots (RODEH, 2007).

[2] Shadow Paging, técnica de gerenciamento de escrita que grava modificações em novas páginas em vez de sobrescrever os blocos originais, atualizando o sistema de forma atômica ao trocar o ponteiro da raiz da árvore (RODEH, 2007).

Os *snapshots* são uma consequência direta dessa arquitetura. Como o estado completo de um subvolume é representado por um único ponteiro raiz, criar um *snapshot* equivale a registrar uma referência adicional para essa raiz em um momento específico. *Snapshot* e subvolume ativo passam a compartilhar todos os nós da árvore existentes no instante da captura. À medida que novas modificações são aplicadas ao subvolume ativo, apenas os nós efetivamente alterados e seus ancestrais recebem cópias novas no disco, enquanto os nós inalterados continuam sendo referenciados por ambas as versões simultaneamente. Configurados com o argumento -r, os snapshots tornam-se somente leitura, o que impede qualquer atualização posterior sobre aquela raiz e garante a imutabilidade do estado registrado. É exatamente esse comportamento que fundamenta a estratégia de versionamento proposta neste estudo, sendo assim, ao gravar no subvolume ativo apenas as *bufferViews* classificadas como modificadas ou novas, e em seguida capturar um *snapshot* imutável, o sistema garante que o custo de armazenamento de cada versão seja proporcional exclusivamente ao volume de dados que efetivamente se alterou.

4.2.5 Método de referência configurado como Git LFS

O método de referência utiliza um repositório Git local, configurado com o Git LFS, na instância *tcc-gitlfs*. Ele funciona através de ganchos que são ativados durante as operações do Git. Estes que, redirecionam os arquivos grandes para um armazenamento especial, fora do histórico principal. A configuração para rastrear arquivos GLB com LFS é feita no arquivo *.gitattributes*, onde é estipulado uma regra que indica que todos os arquivos com essa extensão devem ser tratados como objetos grandes.

Cada operação de versionamento começa com a cópia do arquivo GLB modificado para o diretório do repositório. Em seguida, o comando *git add* é executado. Neste ponto, o gatilho do Git LFS entra em ação, ele lê o arquivo GLB completo, calcula seu *hash* e armazena o arquivo binário no diretório *.git/lfs/objects*. Esse diretório é organizado em subdiretórios baseados nos primeiros *bytes* do *hash* para facilitar a localização. No lugar do arquivo GLB original, o Git armazena apenas um pequeno arquivo de ponteiro, com cerca de 130 *bytes*. Este ponteiro contém informações como o algoritmo de *hash* usado, o identificador do objeto LFS e o tamanho original do arquivo. É esse arquivo de ponteiro que é, de fato, versionado pelo Git, enquanto o conteúdo real do arquivo grande permanece no armazenamento do LFS.

A etapa seguinte é o *git commit*, que registra o arquivo de ponteiro no histórico do repositório, como faria com qualquer outro arquivo de texto. No entanto, o Git LFS não utiliza conceitos como subvolumes, *snapshots* ou compartilhamento de blocos entre versões, que são característicos de sistemas como as Btrfs. Cada objeto armazenado no LFS é um arquivo binário completo e independente. A única forma de economia de espaço é a deduplicação por *hash*, que acontece quando duas versões de um arquivo resultam no mesmo hash, o LFS não cria um segundo objeto. Fora essa situação, que ocorre apenas quando o arquivo não sofreu nenhuma alteração, cada nova versão adiciona uma cópia integral do arquivo ao armazenamento do LFS, independentemente da quantidade de dados que foi modificada em relação à versão anterior.

Para este experimento, isso significa que cada um dos nove cenários de teste, mesmo com pequenas modificações, resulta na adição de aproximadamente 42,9 MB ao diretório *.git/lfs/objects*. O crescimento do armazenamento do LFS é, portanto, linear em relação ao número de versões registradas, sem considerar o volume de dados que realmente mudou.

4.2.6 Métricas de avaliação

Para comparar o desempenho do método proposto com o método de referência, utiliza-se métricas que medem o impacto direto das operações de versionamento no disco. Essas métricas são coletadas de forma padronizada em ambos os ambientes, permitindo uma comparação justa e direta, independentemente de como cada método funciona internamente.

A primeira métrica é a **quantidade de bytes gravados por versão**. Ela representa o volume de dados que é efetivamente escrito no disco quando uma nova versão é registrada. No método proposto, esse valor é a soma dos tamanhos de todas as *bufferViews* que foram classificadas como novas ou modificadas. Matematicamente, isso pode ser expresso como:

$$B\_gravados = \Sigma\, tamanho(bv\_i),\ para\ todo\ bv\_i\ novo\ ou\ modificado \qquad (1)$$

Já no método de referência, usando o Git LFS, essa métrica corresponde ao tamanho do novo objeto binário que é adicionado ao diretório de armazenamento das versões. Isso é medido pela diferença no tamanho total desse armazenamento antes e depois de cada *commit*. Em ambos os casos, essa métrica nos mostra o custo imediato de armazenamento para cada operação de versionamento.

A segunda métrica é a **economia percentual**. Ela mede o quanto de dados não precisou ser gravado em disco em relação ao tamanho total do arquivo original. É expressa em porcentagem e calculada da seguinte forma:

$$E = (1 - B\_gravados / B\_total) * 100 \quad (2)$$

Onde *B_total* é o tamanho completo do arquivo GLB. Esta métrica é importante porque quantifica o ganho de eficiência do método proposto. No Git LFS, o valor de *B_gravados* geralmente é muito próximo de *B_total*, resultando em uma economia percentual próxima de zero. No método proposto, presume-se que a economia aumente à medida que a quantidade de modificações no arquivo diminui, pois mais blocos de dados podem ser compartilhados entre as versões.

A terceira métrica é o **armazenamento acumulado**. Ela representa o espaço total ocupado no disco após o registro de todas as versões ao longo do experimento.

$$A\_acumulado = \Sigma B\_gravados(v\_i),\ para\ i\ de\ 1\ até\ N \quad (3)$$

No método proposto, esse valor é obtido usando o comando btrfs *filesystem usage*, que informa o espaço físico real ocupado no volume, já considerando o compartilhamento de blocos entre os snapshots. No método de referência, é o tamanho total do diretório que armazena os arquivos binários. Embora sejam coletados de locais diferentes, ambos os valores representam a mesma coisa, o espaço total em disco usado pelo histórico completo de versões do arquivo. A comparação entre eles é direta, e a diferença esperada entre os dois métodos mostrará a capacidade do método proposto de reutilizar blocos físicos entre versões consecutivas, gerando economia de espaço.

4.3 Usando o método em áreas individuais

Este segundo experimento aprofunda a avaliação do mecanismo de versionamento diferencial, focando na variação da quantidade de *bufferViews* modificadas, em contraste com a variação da intensidade numérica em todas as *bufferViews* de geometria realizada no primeiro experimento e o modelo usado para a metodologia deste trabalho. O objetivo é analisar o impacto da granularidade das alterações no consumo de armazenamento,

novamente em comparação com o Git LFS. Para isso, o modelo *ABeautifulGame.glb*, composto por 75 *bufferViews* de atributos de vértice, foi submetido a uma série de novas modificações controladas.

Os cenários de teste foram definidos para simular um espectro de alterações, desde modificações mínimas até uma deformação generalizada do modelo. A Tabela 2 apresenta a progressão das versões. Temos uma baixa mudança dentro de cada pedaço, que garante que as deformações visuais sejam sutis, mas suficientes para alterar o hash SHA-256 de cada bufferView modificada, acionando o mecanismo diferencial do sistema. As *bufferViews* de índices de face são preservadas intactas, evitando a corrupção de dados.

Tabela 2 – Cenários de teste e progressão de modificação por *bufferView* em áreas individuais.

Versão	Porcentagem modificada	Quantidade modificada
v1	linha de base	0
v2	1%	1
v3	5%	4
v4	10%	8
v5	25%	19
v6	50%	38
v7	75%	57
v8	100%	75

Fonte: Autoria própria (2026).

Para a implementação deste experimento, o método para gerar as modificações nos arquivos foi adaptado com duas mudanças fundamentais. Primeiramente, foi implementada a separação das *bufferViews* de índices dos atributos, garantindo que apenas os dados de ponto flutuante sejam modificados, evitando a corrupção de índices de face. Em segundo lugar, o script passou a aceitar a quantidade de *bufferViews* a modificar como parâmetro, em vez de um tipo genérico de modificação. Essas adaptações permitem um controle preciso sobre a

granularidade das alterações, essencial para a validação da eficiência do versionamento por blocos em termos de economia de armazenamento, especialmente quando comparado a métodos que não exploram a estrutura interna dos arquivos GLB.

4.4 Cronograma de desenvolvimento

O desenvolvimento do trabalho será organizado em sprints de duas semanas, seguindo os princípios de desenvolvimento iterativo, nos quais cada ciclo possui um objetivo claro e entregável. O período de execução compreende os meses de agosto a novembro de 2026, totalizando sete *sprints*.

1. Agosto, semanas 1 e 2: A primeira *sprint* tem como objetivo a consolidação da arquitetura de versionamento. Serão implementadas as operações fundamentais de recuperação automática de versões anteriores e o registro estruturado do histórico, substituindo a interação direta com o terminal por uma interface programática mais acessível e reproduzível.
2. Agosto, semanas 3 e 4: A segunda *sprint* será dedicada ao estudo e à prototipação de estratégias de fusão entre versões. Serão analisadas as possibilidades de combinar *bufferViews* modificadas em paralelo, identificando cenários de conflito e definindo o comportamento esperado do sistema diante dessas situações.
3. Setembro, semanas 1 e 2: A terceira *sprint* concentra-se na integração do método proposto com um *software* de edição tridimensional. O objetivo é desenvolver um mecanismo que permita acionar o versionamento diretamente a partir do fluxo de trabalho do desenvolvedor, sem necessidade de interação manual com o terminal, tornando a solução utilizável em um ambiente de produção real.
4. Setembro, semanas 3 e 4: A quarta *sprint* abrange o estudo de viabilidade e a prototipação inicial de uma camada *web* sobre a base de versionamento existente. Serão definidas a arquitetura de requisições e as operações básicas que permitirão ao sistema operar de forma distribuída, saindo do repositório estritamente local.
5. Outubro, semanas 1 e 2: A quinta *sprint* dará continuidade ao desenvolvimento do protótipo *web*, consolidando as operações de envio, recuperação e listagem de versões. Ao término desta *sprint*, espera-se ter validado a viabilidade técnica da abordagem distribuída.
6. Outubro, semanas 3 e 4: A sexta sprint será destinada à execução dos experimentos finais e à coleta dos resultados com a arquitetura completa. Os cenários de teste serão

repetidos considerando as novas funcionalidades implementadas, e os dados obtidos fundamentarão a elaboração da seção de resultados e da discussão final do trabalho.

7. Novembro, semanas 1 e 2: A sétima e última *sprint* compreende a revisão geral do texto, a escrita da conclusão, a elaboração da documentação técnica da arquitetura final e a preparação da apresentação para a banca avaliadora.

5 RESULTADOS PARCIAIS

Os resultados apresentados a seguir foram coletados após executar todo o ciclo experimental nas duas máquinas virtuais, cobrindo os oito cenários de teste da Tabela 1. A primeira versão é a base, onde o modelo *ABeautifulGame* é salvo sem nenhuma mudança. As versões seguintes mostram modificações progressivas aplicadas apenas aos dados de geometria, variando de 2% a 90% das *bufferViews*.

No método de referência, usando o Git LFS, observa-se que aproximadamente 42,9 MB foram adicionados ao armazenamento no Git LFS a cada nova versão, sem importar o quanto a modificação era grande. Ao final das oito versões, o armazenamento no Git LFS acumulou 343,9 MB de dados. Isso significa que 8 cópias completas do arquivo foram guardadas. Esse comportamento já era esperado pela forma como o Git LFS funciona: como ele trata o arquivo GLB como um bloco único, qualquer mudança no seu conteúdo gera um identificador único diferente. Isso força o sistema a guardar um novo arquivo completo, mesmo que a alteração tenha sido pequena.

Já no método proposto, o comportamento foi diferente. A versão 1, que é a primeira vez em que o modelo é salvo, precisou gravar todas as 108 *bufferViews*, totalizando 42,9 MB, pois não havia uma versão anterior para comparar. A partir da versão 2, o sistema percebeu que 75 *bufferViews* tinham sido modificadas e 33 continuavam iguais. Isso resultou na gravação de 24,0 MB por versão e uma economia de 44,08% em relação ao tamanho total do arquivo. Esse padrão se manteve em todas as versões seguintes. Isso acontece porque a detecção de diferenças funciona comparando os identificadores únicos das *bufferViews* entre as versões e qualquer mudança nos *bytes* de uma *bufferView*, por menor que seja, muda seu identificador e a marca como modificada. As *bufferViews* que não foram alteradas são ignoradas no processo de escrita. Como os oito cenários de teste sempre modificam o mesmo grupo de 75 *bufferViews* de geometria, variando apenas a intensidade das mudanças

numéricas, o número de segmentos gravados permaneceu o mesmo entre as versões. A Tabela 3 resume os resultados coletados em ambas as instâncias durante o experimento.

Tabela 3 – Comparativo de custos de armazenamento por versão.

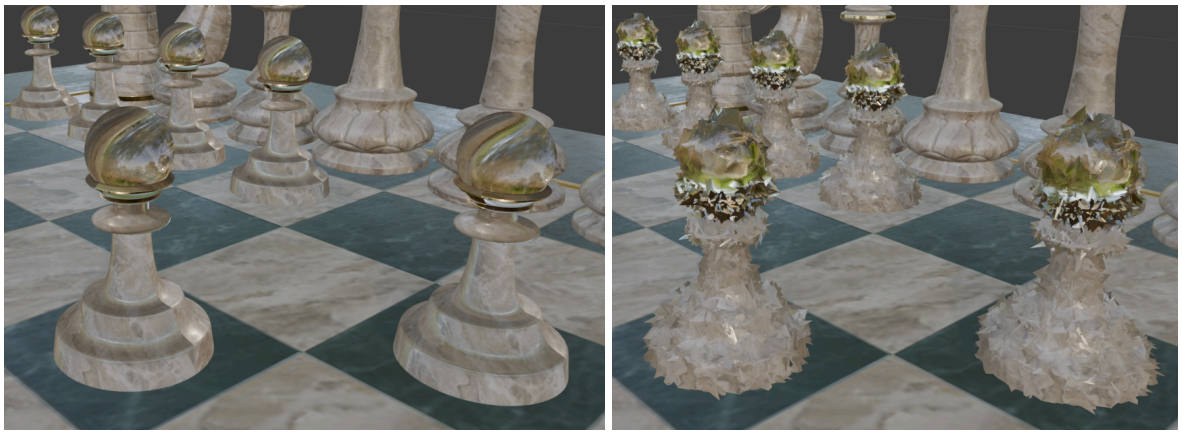
Versão	Modificação (%)	Btrfs em bytes gravados (MB)	Btrfs economia (%)	Git LFS em bytes gravados (MB)
1	0	42	0,00	42
2	2	24	44,08	42
3	5	24	44,08	42
4	10	24	44,08	42
5	20	24	44,08	42
6	30	24	44,08	42
7	50	24	44,08	42
8	70	24	44,08	42
Total	-	**210**	-	**336**

Fonte: Autoria própria (2026).

Ao final das oito versões, o armazenamento acumulado foi de cerca de 211 MB no método proposto, contra 344 MB no Git LFS. Isso representa uma redução de 38,6% no uso total de espaço em disco. É importante notar que a economia do método proposto pode ser ainda maior em situações onde as mudanças atingem um número menor de *bufferViews*. Isso acontece porque a precisão da detecção de mudanças depende da estrutura interna do arquivo, e não da quantidade total de dados que foi alterada.

Sobre o segundo método experimental proposto analisa-se os seguintes resultados. No método de referência, utilizando o Git LFS, observou-se um comportamento consistente de armazenamento. A cada nova versão, aproximadamente 42,9 MB foram adicionados ao repositório, independentemente da quantidade de *bufferViews* modificadas. Ao final das oito versões, o armazenamento total acumulado no Git LFS atingiu cerca de 343,9 MB. A Figura 13 ilustra visualmente a diferença entre o modelo original e o modelo na versão 8, onde todas as *bufferViews* de atributos foram modificadas, resultando em extrusões nos vértices, que, embora sutis, foram suficientes para acionar o mecanismo de versionamento.

Figura 13 - O modelo modificado.



Fonte: Autoria própria (2026).

Em contraste, o método proposto, baseado no Btrfs, demonstrou uma eficiência de armazenamento significativamente superior. A versão inicial registrou 42,9 MB, pois todas as 108 *bufferViews* foram consideradas novas. A partir da segunda versão, o sistema identificou as *bufferViews* modificadas e gravou apenas os dados correspondentes a essas alterações. Por exemplo, na segunda versão, com apenas uma *bufferView* modificada, foram gravados apenas 0,75 MB, resultando em uma economia de 98,25% em relação ao tamanho total do arquivo. Essa economia diminuiu progressivamente à medida que mais *bufferViews* foram modificadas, atingindo 9,31% na oitava versão, onde 93 *bufferViews* foram alteradas, com 38,9 MB gravados. Este comportamento reflete a capacidade do Btrfs de armazenar diferencialmente apenas os blocos de dados que sofreram alterações, compartilhando os blocos inalterados entre as versões por meio de snapshots CoW. Ao final das oito versões, o armazenamento acumulado pelo método proposto foi de 211 MB, representando uma redução de 38,6% no uso total de espaço em disco em comparação com o Git LFS. A Tabela 4 abaixo resume os resultados comparativos de armazenamento para ambos os métodos.

Tabela 4 – Comparativo de custos de armazenamento.

Versão	Btrfs em bytes gravados	Btrfs economia (%)	Git LFS em bytes gravados
1	42.945.647	0,00	42.945.647
2	750.313	98.25	42.945.647
3	1.902.849	95.57	42.945.647
4	3.892.967	90.94	42.945.647
5	10.973.923	74.45	42.945.647
6	19.098.856	55.53	42.945.647
7	28.555.173	33.51	42.945.647
8	38.947.439	9.31	42.945.647
Total	**147.067.167**	-	**343565176**

Fonte: Autoria própria (2026).

6 CONSIDERAÇÕES FINAIS

Este trabalho investigou as limitações dos mecanismos tradicionais de versionamento no tratamento de arquivos binários, com ênfase no contexto do desenvolvimento de jogos digitais. A análise evidenciou que ferramentas consolidadas como o Git LFS, embora amplamente adotadas na indústria, tratam arquivos binários como unidades opacas e indivisíveis, resultando no armazenamento integral do arquivo a cada nova versão. A identificação do formato GLB como objeto de estudo adequado demonstrou que formatos binários com estrutura interna segmentada, *offsets* explícitos e ausência de compressão global apresentam características que viabilizam o versionamento diferencial por blocos, distinguindo-se de formatos que tornaram essa abordagem ineficiente, como arquivos compactados.

A investigação das técnicas CoW e RoW, bem como do sistema de arquivos Btrfs, permitiu compreender como essas abordagens podem ser aplicadas ao problema do versionamento diferencial de arquivos binários. A arquitetura de árvores-B do Btrfs, combinada ao mecanismo de *shadow paging*, mostrou-se adequada para o problema proposto, onde a criação de um snapshot equivale ao registro de um novo ponteiro raiz, operação de custo constante independente do volume de dados, e o compartilhamento físico de blocos inalterados entre versões consecutivas ocorre de forma nativa, sem duplicação em disco.

Esses conceitos fundamentaram as decisões de projeto da camada física do método desenvolvido.

A abordagem proposta foi desenvolvida em duas camadas complementares. Na camada lógica, um módulo implementado realiza a leitura e decomposição do arquivo GLB, extrai as *bufferViews* como unidades de versionamento, calcula assinaturas por segmento e executa a detecção diferencial entre versões consecutivas, encaminhando para gravação apenas os segmentos classificados como novos ou modificados. Na camada física, o Btrfs recebe esses segmentos no subvolume ativo e, ao término de cada operação, captura um snapshot imutável que preserva o estado completo da versão com custo proporcional exclusivamente ao volume de dados alterado.

A avaliação comparativa com o Git LFS, conduzida em ambientes virtualizados isolados com configurações de *hardware* idênticas, produziu resultados parciais que validam a viabilidade da abordagem. No experimento com variação da quantidade de *bufferViews* modificadas, o método proposto alcançou economias que variaram de 98,25%, quando apenas uma bufferView foi alterada, a 9,31% no cenário de modificação integral dos atributos de vértice, enquanto o Git LFS manteve crescimento constante de aproximadamente 42,9 MB por versão em todos os cenários. Ao final de oito versões, o armazenamento acumulado foi de aproximadamente 211 MB no método proposto contra 344 MB no Git LFS, representando uma redução de 38,6% no consumo total de espaço em disco.

Os resultados parciais obtidos são promissores e permitem especular aplicações concretas do método em diferentes contextos. Na indústria de jogos digitais por exemplo, estúdios que mantêm repositórios com centenas de modelos GLB modificados diariamente por equipes de artistas se beneficiariam diretamente ao ver o cenário onde uma alteração pequena gravou apenas 750 KB em vez de 42,9 MB, uma economia de 98%. Times com dezenas de artistas realizando *commits* simultâneos representam uma diferença expressiva no consumo acumulado de banda e armazenamento ao longo do ciclo de produção. Em pipelines de integração e entrega contínua para ativos 3D, onde versões de modelos precisam ser validadas automaticamente a cada alteração, apenas os segmentos modificados precisariam ser transferidos entre máquinas, reduzindo o tempo de propagação. Em ambientes de colaboração remota com largura de banda limitada, a diferença entre transferir 750 KB e 42 MB por *commit* representa a viabilidade ou inviabilidade do fluxo de trabalho. Por fim, o método proposto não busca se restringir ao formato GLB, em estudos futuros seram levantados métodos para atender a qualquer formato binário com estrutura interna segmentada e *offsets* explícitos, estes que apresentam as condições necessárias para sua aplicação.

REFERÊNCIAS

ALDUS CORPORATION. **TIFF Revision 6.0**. Seattle, WA: Aldus Corporation, 1992. Disponível em: https://www.itu.int/itudoc/itu-t/com16/tiff-fx/docs/tiff6.pdf. Acesso em: 09 maio 2026.

BENITES, Vitor de Freitas; FLORES, Gustavo Lopez; RODRIGUES, Fabio Batista. **Basic Solution File System.** Faculdade de Computação, Universidade Federal de Mato Grosso do Sul, Campo Grande, 2025.

CANONICAL. **Multipass documentation**. [S.l.]: Canonical, 2026. Disponível em: https://documentation.ubuntu.com/multipass/stable. Acesso em: 09 maio 2026.

CHEN, Hsiang-Ting; WEI, Li-Yi; CHANG, Chun-Fa. **Nonlinear revision control for images.** ACM Transactions on Graphics, New York, v. 30, n. 4, art. 105, p. 1-10, jul. 2011.

CHEN, Jie; WANG, Jun; TAN, Zhihu; XIE, Changsheng. **Effects of Recursive Update in Copy-on-Write File Systems: A BTRFS Case Study.** Canadian Journal of Electrical and Computer Engineering, v. 37, p. 113–120, 2014.

CHEN, Yuan; WANG, Qinying; YANG, Yong; CHEN, Yuanchao; LI, Yuwei; JI, Shouling. **Unveiling security vulnerabilities in Git Large File Storage protocol.** In: IEEE SYMPOSIUM ON SECURITY AND PRIVACY (SP), 2025.

CUNHA, M. B. **Entendendo o uso do Git em equipes de desenvolvimento de software.** Recife – Brasil. UFPE, 2018. Disponível em:

DOBOŠ, Jozef; STEED, Anthony. **3D revision control framework**. In: INTERNATIONAL CONFERENCE ON 3D WEB TECHNOLOGY (Web3D). Los Angeles. New York, 2012.

HOUSTON, Ben. **glTF: everything you need to know.** Threekit Blog, 18 abr. 2019. Disponível em: https://www.threekit.com/blog/gltf-everything-you-need-to-know. Acesso em: 28 mar. 2026.

IBM CORPORATION; MICROSOFT CORPORATION. **Multimedia Programming Interface and Data Specifications 1.0.** [S.l.]: IBM/Microsoft, 1991. p. 3-22 - 3-31. Acesso em: 09 maio 2026.

KASAMPALIS, Sakis. **Copy On Write Based File Systems Performance Analysis And Implementation**. 2010. 83 f. Department of Informatics, Technical University of Denmark, Kongens Lyngby, 2010.

KHRONOS GROUP. **glTF 2.0 Reference Guide.** [S.l.]: Khronos Group, 2022. Disponível em: https://www.khronos.org/files/gltf20-reference-guide.pdf. Acesso em: 28 mar. 2026.

KHRONOS GROUP. **glTF 2.0 Specification. Version 2.0.1.** Beaverton: Khronos Group, 2021. Disponível em: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.pdf. Acesso em: 28 mar. 2026.

KHRONOS GROUP. **glTF-Sample-Models.** Beaverton: Khronos Group, 2026. Disponível em: https://github.com/KhronosGroup/glTF-Sample-Models/. Acesso em: 26 abr. 2026.

MACDONALD, Joshua P. **File System Support for Delta Compression.** Berkeley: University of California at Berkeley, 2000.

PETERSON, Zachary N. J.; BURNS, Randal. **Ext3cow: a time-shifting file system for regulatory compliance.** ACM Transactions on Storage, New York, 2005. Disponível em: https://www.researchgate.net/publication/220398200_Ext3cow_A_Time-Shifting_File_System_for_Regulatory_Compliance. Acesso em: 7 abr. 2026.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de software: uma abordagem profissional.** Porto Alegre – Brasil. AMGH, 2016.

RODEH, Ohad. **B-trees, shadowing, and clones. In: Linux Storage and File System Workshop**. Proceedings […]. 2007. Disponível em: https://www.usenix.org/legacy/event/lsf07/tech/rodeh.pdf. Acesso em: 5 maio 2026.

SCHELL, J. **The Art of Game Design: A Book of Lenses.** Burlington – USA. Elsevier, 2008.

SPINELLIS, Diomidis. **Version control systems.** IEEE Software, v. 22, n. 5, p. 108–109, set. 2005. Disponível em: https://ieeexplore.ieee.org/document/1504674. Acesso em: 29 mar. 2026.

XIAO, Weijun; YANG, Qing; REN, Jin; XIE, Changsheng; LI, Huaiyang. **Design and analysis of block-level snapshots for data protection and recovery.** IEEE Transactions on Computers, v. 58, [s.l.], IEEE Computer Society, 2009.

ZHENG, S. et al. **HMVFS: A Hybrid Memory Versioning File System.** In: SYMPOSIUM ON MASS STORAGE SYSTEMS AND TECHNOLOGIES (MSST), 32., 2016, Santa Clara, CA. Santa Clara, CA: IEEE, 2016. p. 1–12.