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

A Block-Based Differential Storage Approach for Binary File Versioning

Monografia de Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Tecnologia em Sistemas para Internet do Curso Superior de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná.

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 o Btrfs que, ao registrar uma modificação, preserva os dados anteriores sem sobrescrevê-los, compartilhando os blocos inalterados entre as versões. O método foi implementado na ferramenta gitbin e integrado ao Blender por meio de um *addon*. Os experimentos compararam o método proposto com o Git LFS, solução convencional para armazenar arquivos grandes em repositórios de código, em três modelos tridimensionais submetidos a uma sessão de dez versões de edição no Blender. O método reduziu em 89,2%, em média, o espaço ocupado pelo histórico de versões, e as edições pequenas e localizadas gravaram menos de 0,5% do que o Git LFS gravou para a mesma edição. Em contrapartida, o registro de cada versão foi de 1,6 a 2,5 vezes mais lento.

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. The method was implemented in the gitbin tool and integrated into Blender through an add-on. Experiments compared the proposed method against Git LFS, the conventional solution for storing large files in code repositories, using three three-dimensional models subjected to a ten-version editing session in Blender. The method reduced the storage used by the version history by 89.2% on average, and small, localized edits wrote less than 0.5% of what Git LFS wrote for the same edit. In return, recording each version was 1.6 to 2.5 times slower.

Keywords: binary file versioning; differential storage; block-level deduplication; modern file systems.

LISTA DE FIGURAS

Figura 1 – Estado inicial do arquivo binário dividido em blocos 14

Figura 2 – Disco após as mudanças no arquivo 15

Figura 3 – Arquitetura modular do formato glTF 19

Figura 4 – Divisões internas do arquivo JSON no formato glTF 20

Figura 5 – Representação da hierarquia de camadas em um modelo 21

Figura 6 – Alterações em uma imagem de entrada 23

Figura 7 – Estrutura do histórico de revisões e evolução do modelo 24

Figura 8 – Fluxo de observação do arquivo 26

Figura 9 – Cena tridimensional disponibilizada pelo Khronos Group 27

Figura 10 – Vértices editados, em laranja, na peça do cavalo 37

Figura 11 – Crescimento do repositório a cada versão no *ABeautifulGame* 38

Figura 12 – Crescimento do repositório a cada versão no *Spring* 39

Figura 13 – Crescimento do repositório a cada versão no *San Miguel* 39

Figura 14 – Classificação das *bufferViews* do arquivo a cada versão no *ABeautiful-Game* 40

Figura 15 – Classificação das *bufferViews* do arquivo a cada versão no *Spring* . . . 40

Figura 16 – Classificação das *bufferViews* do arquivo a cada versão no *San Miguel* 41

Figura 17 – Tempo médio do *save* por etapa no *ABeautifulGame* 41

Figura 18 – Tempo médio do *save* por etapa no *Spring* 42

Figura 19 – Tempo médio do *save* por etapa no *San Miguel* 42

Figura 20 – Razão de tempo do *save*, gitbin em relação ao *Git Large File Storage* (Git LFS) 43

Figura 21 – Razão do pico de memória do Blender, gitbin em relação ao Git LFS . . 43

Figura 22 – Razão do tempo de restauração, gitbin em relação ao Git LFS 43

LISTA DE TABELAS

Tabela 1 – Cenários de teste e intensidades de modificação **28**
Tabela 2 – Cenários de teste e progressão de modificação por bufferView em áreas individuais **33**
Tabela 3 – Comandos do gitbin e etapas do método **35**
Tabela 4 – Iterações de melhoria do gitbin e do *addon* **36**
Tabela 5 – Configuração das máquinas virtuais **49**
Tabela 6 – Características dos modelos utilizados **49**
Tabela 7 – Crescimento do repositório a cada versão (MB) **50**
Tabela 8 – Tempo do *save* a cada versão (s) **50**
Tabela 9 – Pico de memória do Blender a cada versão (GB) **51**
Tabela 10 – Tempo de restauração (*checkout*) a cada versão (s) **51**
Tabela 11 – Dados gravados em disco a cada versão, medidos com o `iostat` (MB) **52**
Tabela 12 – Resultado da verificação das versões restauradas **52**

LISTA DE ABREVIATURAS E SIGLAS

Siglas

ASCII	*American Standard Code for Information Interchange*
BSON	*Binary JSON*
Btrfs	*B-tree File System*
CoW	*Copy-on-Write*
DAG	*Directed Acyclic Graph*
Git LFS	*Git Large File Storage*
GLB	*GL Transmission Format Binary*
glTF	*GL Transmission Format*
JSON	*JavaScript Object Notation*
LFS	*Large File Storage*
NoSQL	*Not Only SQL*
PBR	*Physically Based Rendering*
RevG	*Revision Graph*
RoW	*Redirect-on-Write*
SHA-256	*Secure Hash Algorithm 256 bits*
TIFF	*Tagged Image File Format*
UIP	*Update-in-Place*
URI	*Uniform Resource Identifier*
UTF-8	*Unicode Transformation Format 8-bit*
VDB	*Volumetric Dynamic B plus tree*
WAVE	*Waveform Audio File Format*
WebGL	*Web Graphics Library*

XDFS	*Xdelta File System*
ZFS	*Zettabyte File System*

SUMÁRIO

1 INTRODUÇÃO 9

1.1 Objetivos 10

1.1.1 Objetivo geral 10

1.1.2 Objetivos específicos 10

1.2 Justificativa 10

2 REFERENCIAL TEÓRICO 12

2.1 Sistema de controle de versões 12

2.1.1 O Sistema de Controle de Versão Git 12

2.2 Modelos de escrita para sistemas de arquivos 13

2.2.1 *Copy-on-Write* e *Redirect-on-Write* 14

2.3 Sistemas de arquivos modernos usando *Copy-on-Write* (CoW) 15

2.3.1 *B-tree File System* 16

2.4 Arquivos binários 16

2.4.1 Desafios no versionamento de arquivos binários 17

2.4.2 Versionamento baseado em blocos 17

2.4.3 Limitações do versionamento por blocos em arquivos binários 18

2.5 O formato glTF 18

2.5.1 Estrutura geral de um ativo glTF 18

2.5.2 O contêiner GLB 19

2.5.3 Arquivos similares 21

2.6 Trabalhos relacionados 22

2.6.1 Controle de revisão não linear para imagens 22

2.6.2 *Framework* de controle de revisão 3D 23

2.6.3 Suporte a compressão delta em sistemas de arquivos 24

2.6.4 Análise comparativa dos trabalhos relacionados 25

3 METODOLOGIA 26

3.1 Procedimentos Experimentais 26

3.1.1 Objeto de teste 27

3.1.2 Geração controlada das versões 27

3.1.3 Método proposto 28

3.1.4 Conceitos Fundamentais da Camada Física 29
3.1.5 Método de referência configurado como Git LFS 31
3.1.6 Métricas de avaliação 31
3.1.7 Aplicação do método em áreas individuais 32
4 RESULTADOS E DISCUSSÕES 34
4.1 A ferramenta gitbin 34
4.2 Integração com o Blender 34
4.3 Limitação do exportador GLB do Blender 35
4.4 Discussão 37
4.4.1 Custo de cada versão 38
4.4.2 Custo em tempo e memória 40
5 CONSIDERAÇÕES FINAIS 44
REFERÊNCIAS 46
APÊNDICE A DADOS COMPLEMENTARES DOS EXPERIMENTOS FINAIS 49
A.1 Ambiente e modelos 49
A.2 Resultados por versão 49
A.3 Integridade das versões 50

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 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 Objetivos

Os objetivos do trabalho se dividem em objetivo geral e objetivos específicos.

1.1.1 Objetivo geral

Desenvolver um novo mecanismo de versionamento de arquivos binários fundamentado em arquiteturas baseadas nos princípios de 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.

1.1.2 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 CoW e 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.

1.2 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). Apesar de essa abordagem reduzir 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 CoW e RoW, amplamente utilizadas em sistemas de armazenamento para a criação de *snapshots* diferenciais, conforme apresentado por Xiao *et al.* (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 na mesma medida que os arquivos de texto.

2 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.

2.1 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 (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.

2.1.1 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).

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). O 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 *Large File Storage* (LFS) especializado. Com isso, o repositório Git mantém apenas metadados sobre cada arquivo, 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).

2.2 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 utili-

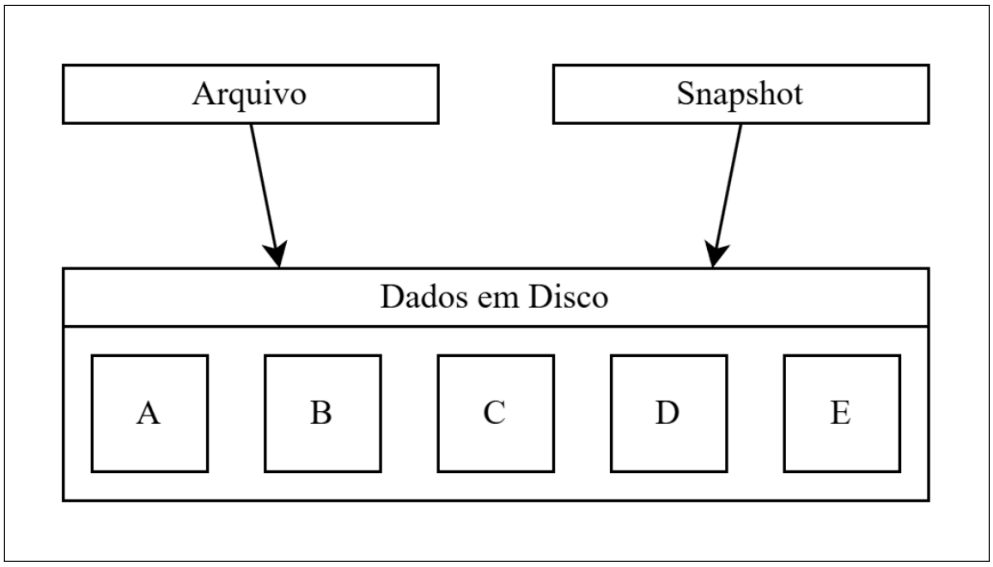
zados para implementar essas estratégias de atualização de dados (XIAO *et al.*, 2009), as quais são utilizadas nos sistemas de arquivos modernos, como Btrfs e ZFS.

2.2.1 *Copy-on-Write* e *Redirect-on-Write*

O modelo 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).

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).

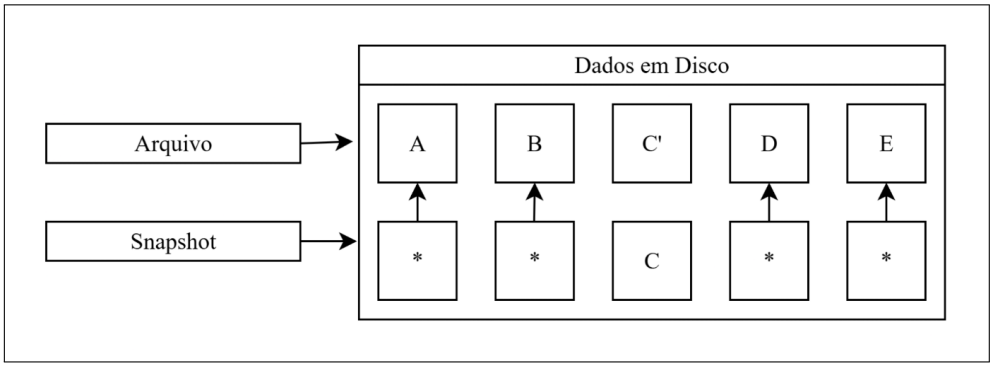
Figura 1 – Estado inicial do arquivo binário dividido em blocos



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. 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 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).

2.3 Sistemas 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).

2.3.1 *B-tree File System*

O 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, o sistema de arquivos 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 (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).

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).

2.4 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. 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).

2.4.1 Desafios no versionamento de arquivos binários

Uma alternativa para lidar com esse problema consiste em utilizar mecanismos de armazenamento baseados em blocos, mencionado anteriormente. 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).

2.4.2 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.

2.4.3 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).

2.5 O formato glTF

O *GL Transmission Format* (glTF) 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 *Web Graphics Library* (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).

2.5.1 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:

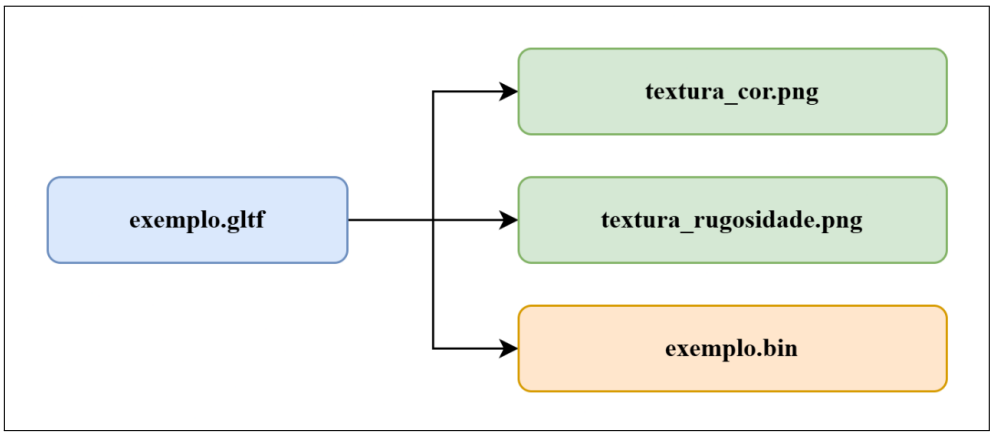
a) O Arquivo de Descrição (.gltf) consiste em um arquivo estruturado em formato *JavaScript Object Notation* (JSON) responsável por representar a estrutura lógica da cena. Nele são definidos elementos como a hierarquia de nós, materiais, câmeras, animações e referências aos demais recursos utilizados 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.

Figura 3 – Arquitetura modular do formato glTF



Fonte: Autoria própria (2026).

2.5.2 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 *Unicode Transformation Format 8-bit* (UTF-8) sem conter a sequência de *bytes* inicial; as chaves de objetos devem ser únicas e nomes de propriedades devem usar apenas caracteres *American Standard Code for Information Interchange* (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;

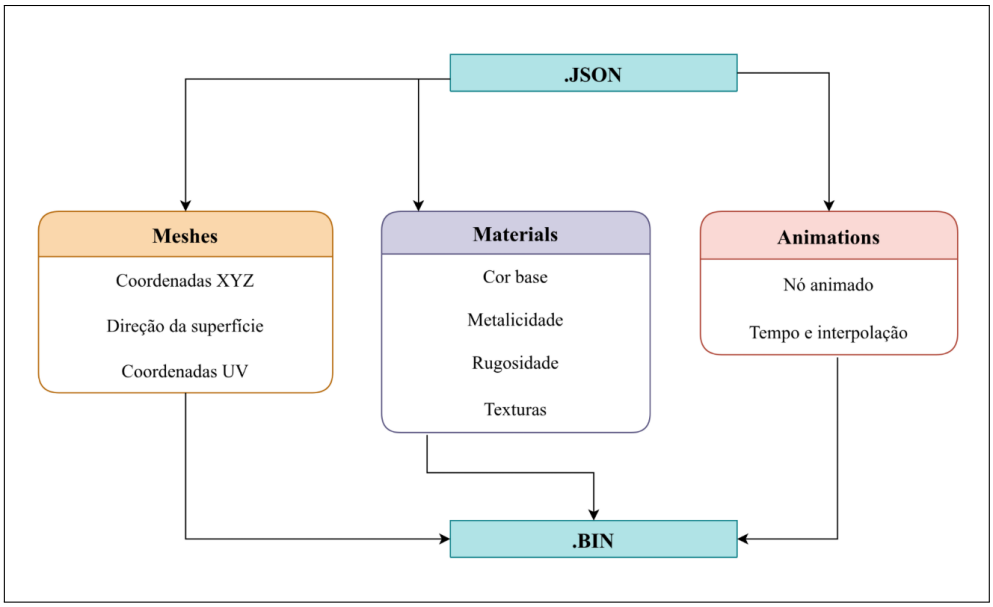
b) Os *materials* especificam as propriedades ópticas dos modelos, incluindo mapeamentos de cor base, metalicidade, rugosidade e texturas de normais;

c) *Animations* operam através de uma estrutura de canais e *samplers*;

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.

Figura 4 – Divisões internas do arquivo JSON no formato glTF



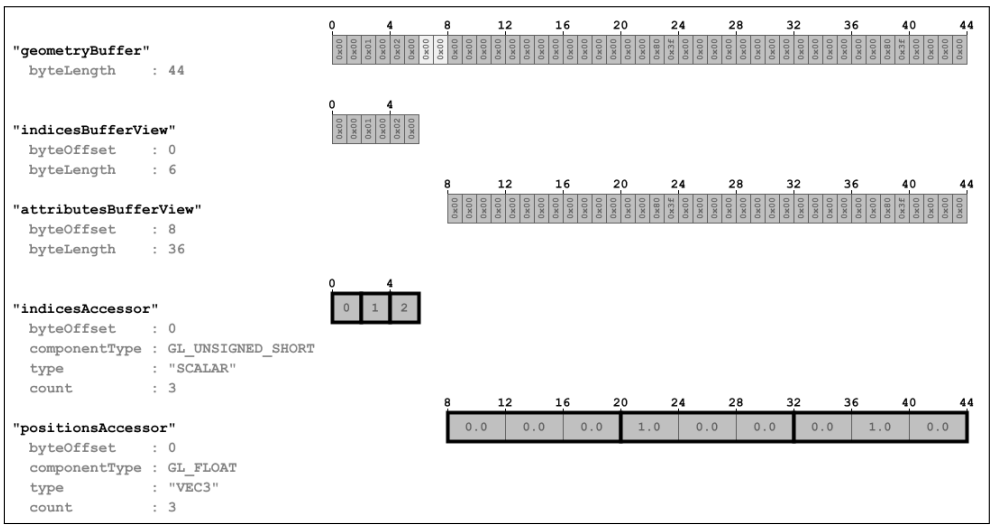
Fonte: Autoria própria (2026).

Uma funcionalidade crítica do JSON é o referenciamento de recursos externos por meio de *Uniform Resource Identifier* (URI), 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.

Figura 5 – Representação da hierarquia de camadas em um modelo



Fonte: KHRONOS GROUP (2022).

2.5.3 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 *Tagged 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, siste-

mas operacionais ou *hardware* específicos, o que o torna possivelmente extensível ao método proposto neste trabalho. 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 vistos 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 de maneira aleatória e eficiente (ALDUS CORPORATION, 1992).

De forma análoga, o formato *Waveform Audio File Format* (WAVE) para áudio, conforme especificações conjuntas da IBM Corporation e da Microsoft Corporation (IBM CORPORATION; 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.

2.6 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.

2.6.1 Controle de revisão não linear para imagens

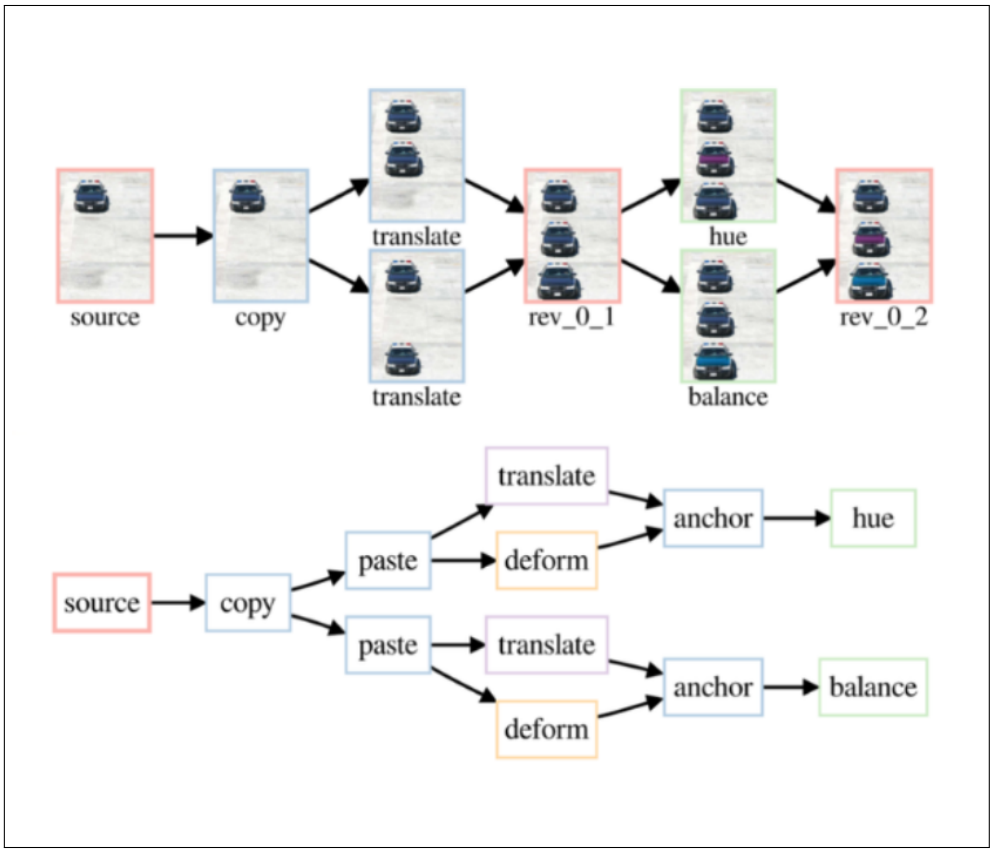
O trabalho desenvolvido por Chen, Wei e Chang (2011) apresenta um sistema de controle de versão para imagens digitais, que se diferencia dos métodos tradicionais ao registrar as ações de edição do usuário em vez de armazenar o arquivo completo a cada versão. Este enfoque visa otimizar o armazenamento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

O funcionamento do sistema baseia-se em duas estruturas primárias, o *Directed Acyclic Graph* (DAG) e o *Revision Graph* (RevG). O DAG constitui o cerne do sistema, no qual 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 (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, trans-

lação, 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).

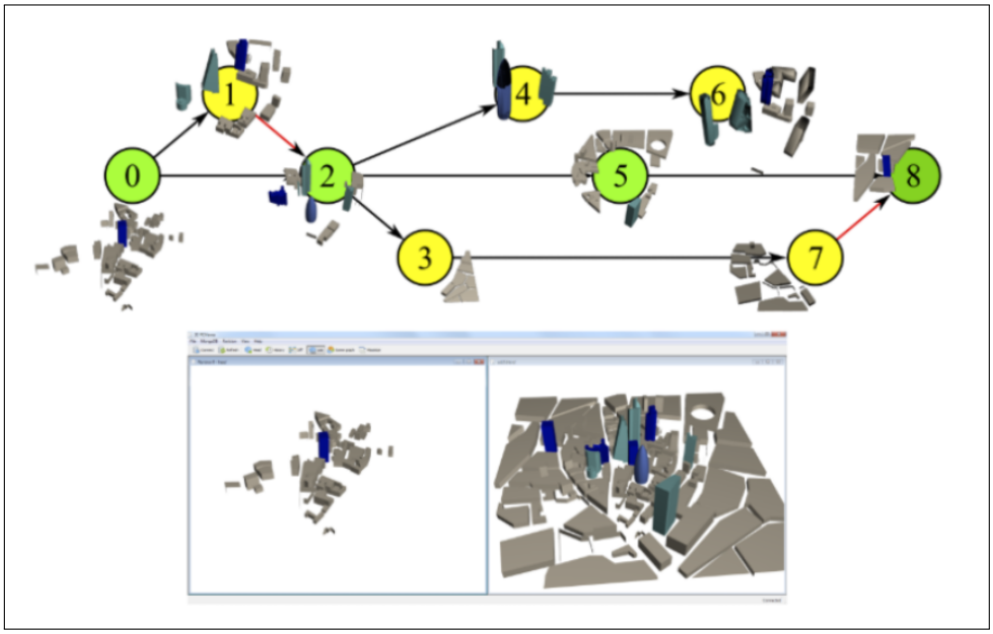
2.6.2 *Framework* de controle de revisão 3D

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 *Not Only SQL* (NoSQL), MongoDB, como repositório central. A cena 3D é desconstruída em um Grafo de Cena, no qual componentes como malhas, materiais e câmeras são armazenados como documentos binários individuais no formato *Binary JSON* (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 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 DAG, no qual

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 (DOBOŠ; STEED, 2012).

Figura 7 – Estrutura do histórico de revisões e evolução do modelo



Fonte: Doboš e Steed (2012).

2.6.3 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.

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 (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, conforme discutido na Seção 2.4.3 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).

2.6.4 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 os autores 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.

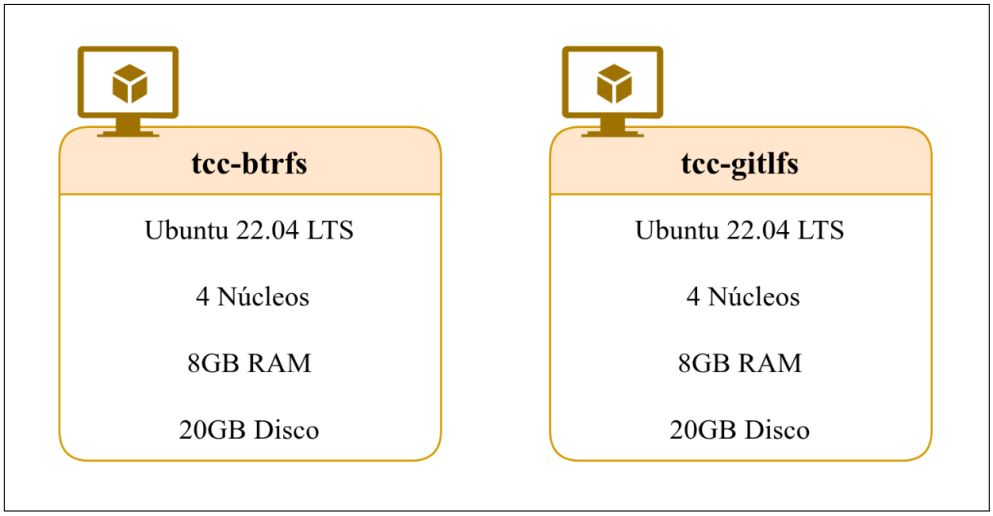
Chen, Wei e Chang (2011) versionam não os dados do arquivo, mas o registro das ações de edição do usuário, o que exige integração direta com o *software* de edição. O presente trabalho atua em uma camada distinta, tendo como objeto de versionamento os próprios *bytes* resultantes da edição, tornando 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.

3 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 *GL Transmission Format Binary* (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 é explicada mais detalhadamente ao longo desta seção.

Figura 8 – Fluxo de observação do arquivo



Fonte: Autoria própria (2026).

Para a implementação do algoritmo de segmentação e análise dos arquivos GLB, utiliza-se 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 é 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 provêm do repositório oficial de amostras do Khronos Group (KHRONOS GROUP, 2026).

3.1 Procedimentos Experimentais

Os procedimentos experimentais são organizados de forma a permitir uma comparação direta e controlada entre a abordagem proposta e o método de referência. Para isso, ambos os

ambientes recebem o mesmo modelo tridimensional como entrada e são submetidos ao mesmo conjunto de versões modificadas, geradas com parâmetros idênticos e reproduzíveis.

3.1.1 Objeto de teste

A escolha do modelo tridimensional `ABeautifulGame.glb`[1] 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 2.4.2 do referencial teórico. O modelo atende aos critérios de um *layout* previsível e de uma segmentação interna bem definida por *bufferViews*, característica essencial para a aplicação eficiente de estratégias de armazenamento diferencial. Adicionalmente, o modelo não possui compressão global nem criptografia, fatores que comprometem a granularidade e a eficácia do versionamento baseado em blocos (MACDONALD, 2000). O modelo selecionado pode ser observado na Figura 9.

Figura 9 – Cena tridimensional disponibilizada pelo Khronos Group



Fonte: KHRONOS GROUP (2026).

3.1.2 Geração controlada das versões

Para a criação das versões modificadas do objeto de teste, utiliza-se um *script* de modificação especificamente para simular edições típicas realizadas por um artista sobre a geometria de um modelo. As alterações são 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 ocorrem em um fluxo de trabalho de modelagem tridimensional.

A reprodutibilidade dos experimentos é assegurada pela utilização de uma semente fixa para o gerador de números pseudoaleatórios. 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.

[1] MACKEY, Ed; MATERIALX PROJECT. *A Beautiful Game*. In: KHRONOS GROUP. *glTF Sample Assets*. Disponível em: https://github.khronos.org/glTF-Assets/model/ABeautifulGame. Acesso em: 11 nov. 2026.

O primeiro experimento considera 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).

3.1.3 Método proposto

O mecanismo de versionamento desenvolvido possui duas camadas complementares. Na camada lógica, ilustrada na Figura 8, um módulo de análise implementado em Python realiza a leitura e a decomposição do arquivo GLB. Na camada física, o sistema de arquivos Btrfs é responsável pelo armazenamento diferencial e pela preservação das versões anteriores por meio de *snapshots* baseados no mecanismo CoW.

A árvore correspondente ao primeiro *snapshot* é composta pela raiz inicial, pelos nós internos e pelos nós folha contendo as *bufferViews* originais. Após o registro de uma nova versão contendo uma modificação, quando o sistema detecta a alteração de uma *bufferView*, como por exemplo a `bv_0011`, 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 do segundo *snapshot*.

É importante destacar que isso não representa a criação de uma segunda árvore completa: apenas os nós no caminho entre o nó modificado e a raiz recebem cópias novas. Todos os demais nós continuam existindo uma única vez no disco, compartilhados entre os dois *snapshots* sem qualquer duplicação. 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*.

Após a compreensão do funcionamento das duas camadas, definem-se as etapas gerais do método. A primeira etapa consiste na leitura do contêiner GLB e em sua separação em pedaços. Conforme descrito na Seção 2.5.2, o formato GLB organiza seu conteúdo em um JSON, responsável pela descrição da estrutura da cena, e um arquivo binário, que armazena os dados brutos. O módulo de análise lê o cabeçalho de 12 *bytes* do arquivo, identifica as porções do arquivo por seus identificadores de tipo e as separa em memória para processamento nas etapas seguintes.

A segunda etapa consiste na extração das *bufferViews* como unidades de versionamento. Como apresentado na Seção 2.5.2, 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.

O terceiro passo envolve a criação de uma assinatura digital única para cada *bufferView*. O algoritmo *Secure Hash Algorithm 256 bits* (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 da versão anterior. Nesse sentido, cada *bufferView* é classificada em uma de quatro categorias: nova, modificada, removida ou inalterada. Apenas as 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 dos segmentos modificados, o mecanismo nativo de *snapshots* do Btrfs é acionado para registrar o estado atual do subvolume como uma versão imutável de leitura.

3.1.4 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 atual, 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.

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 (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 gra-

[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).

var 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 altera.

3.1.5 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. Esses ganchos 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 estipulada 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 o 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 é modificada em relação à versão anterior.

3.1.6 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.

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:

$$B_{gravados} = \sum tamanho(bv_i), \text{ para todo } bv_i \text{ novo ou modificado} \quad (1)$$

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*.

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:

$$E = \left(1 - \frac{B_{gravados}}{B_{total}}\right) \times 100 \quad (2)$$

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.

$$A_{acumulado} = \sum_{i=1}^{N} B_{gravados}(v_i) \quad (3)$$

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.

3.1.7 Aplicação do método em áreas individuais

O 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 empregada no primeiro experimento. O objetivo foi 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, é submetido a uma série de novas modificações controladas.

Os cenários de teste são 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 é adaptado com duas mudanças fundamentais. Primeiramente, é 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* passa a aceitar a quantidade de *bufferViews* a modificar como parâmetro, em vez de um tipo genérico de modificação.

4 RESULTADOS E DISCUSSÕES

Este capítulo apresenta a materialização do método em uma ferramenta de linha de comando e sua integração a um *software* de modelagem tridimensional, bem como as limitações observadas durante os testes.

4.1 A ferramenta gitbin

Em um primeiro momento, para a experimentação e a validação do método, foi utilizado um conjunto de *scripts* em Python, entre eles o `glb_parser.py`, responsável por ler e separar o arquivo GLB, e o `modificar_glb.py`, que gerava versões com alterações controladas do modelo. Esses programas precisavam ser chamados manualmente e em uma sequência específica, de modo que era necessário conhecer a ordem e os argumentos de cada um. Antes mesmo dessa execução, também era preciso preparar o ambiente por meio de comandos como `mkfs.btrfs` e `mount -o loop`, utilizados para criar e montar a imagem do sistema de arquivos, além de criar os subvolumes e os *snapshots*. Como o fluxo envolvia dezenas de etapas encadeadas, um erro em qualquer ponto do processo exigia a reconfiguração completa do ambiente.

Além disso, os caminhos utilizados eram fixos e escritos diretamente nos *scripts* e nos comandos de terminal, como `/mnt/experimento` e `~/tcc/resultados`, o que atendia ao ambiente controlado das máquinas virtuais, mas limitava o uso do método a esse cenário. As operações também seguiam um único sentido, o que tornava trabalhoso voltar ou editar versões passadas.

Diante desses problemas, enfrentados durante a definição do método, desenvolveu-se o gitbin, uma ferramenta de linha de comando instalável via `pip` que empacota o método de versionamento diferencial em uma interface programática, eliminando a necessidade de interação direta com os comandos do Btrfs e com *scripts* separados. A ferramenta está disponível sob licença aberta[1]. Seguindo a mesma lógica de uso do Git, o gitbin implementa cinco comandos, descritos na Tabela 3, e cada um deles abstrai internamente as etapas do método, como a leitura do GLB, a extração das *bufferViews*, o cálculo dos *hashes* SHA-256, a detecção diferencial e a criação dos *snapshots* imutáveis.

4.2 Integração com o Blender

Mesmo com o gitbin, o fluxo de trabalho de um artista tridimensional ainda exigia alternar entre o editor e o terminal, era necessário exportar manualmente o modelo para o formato GLB, abrir o terminal, navegar até o diretório do repositório e só então executar o comando de registro

[1] https://github.com/FelipeFadel/gitbin

Tabela 3 – Comandos do gitbin e etapas do método

Comando	Função	Etapa do método
`gitbin init`	Inicializa o repositório com uma imagem Btrfs configurável	Camada física
`gitbin save`	Registra uma nova versão	Etapas 1 a 5
`gitbin status`	Verifica as alterações pendentes	Etapas 1 a 4
`gitbin log`	Lista o histórico de versões	*Snapshots*
`gitbin checkout`	Restaura uma versão anterior	*Snapshots*

Fonte: Autoria própria (2026).

da versão. Essa troca constante de contexto representa um atrito no processo de modelagem e aumenta a chance de que versões deixem de ser registradas. Diante disso, foi desenvolvido um *addon* para o Blender, na versão 5.2.2, que integra os comandos do gitbin diretamente ao editor tridimensional.

O *addon* aciona o gitbin como processo externo, executando os mesmos comandos disponíveis no terminal e exibindo sua saída no próprio painel. O repositório utilizado é o diretório `.gitbin` localizado na mesma pasta do arquivo GLB informado no painel, o que exige que o repositório tenha sido previamente inicializado com o comando `gitbin init`.

O fluxo começa pelo botão *Importar*, que abre o modelo em uma cena limpa e prepara cada malha para a detecção de alterações, mecanismo detalhado na Seção 4.3. Ao clicar em *Save Version*, o *addon* aciona o exportador GLB do Blender, combina o resultado com a versão anterior e, em seguida, executa o comando `gitbin save`, materializando em um único clique o fluxo completo de exportação e versionamento. O *Checkout*, por sua vez, restaura a versão informada e reimporta o arquivo resultante na cena, permitindo que o artista visualize imediatamente o estado anterior do modelo. O *addon* avisa sobre alterações não salvas antes de restaurar uma versão. Dessa forma, o método passou a ser utilizável em um fluxo real de modelagem, sem que o artista precise sair do editor.

Durante os testes dessa integração, entretanto, foi identificada uma limitação relevante no comportamento do exportador do Blender, descrita na seção seguinte.

4.3 Limitação do exportador GLB do Blender

Os primeiros testes do *addon* mostraram que abrir o modelo *ABeautifulGame* no Blender e reexportá-lo sem nenhuma edição levava o gitbin a gravar 33,30 MB, quase o arquivo inteiro. A investigação apontou duas causas somadas, uma no exportador e outra na própria ferramenta.

A primeira causa encontrada é que o exportador GLB do Blender não preserva a estrutura original do arquivo importado, assim a cada exportação, a geometria é reconstruída a partir da representação interna do editor. Nessa reconstrução, a quantidade de vértices muda, a or-

dem dos vértices e dos índices é alterada, as normais sofrem arredondamentos de até 0,0006, as tangentes são descartadas e as malhas são reordenadas por nome, o que reduziu o arquivo de 108 para 86 *bufferViews*.

A segunda causa estava na detecção diferencial do gitbin, que comparava cada *bufferView* com a de mesma posição na versão anterior. Quando uma *bufferView* era inserida ou removida, todas as seguintes mudavam de índice e eram tratadas como modificadas, ainda que seu conteúdo fosse idêntico. Em um dos testes, mover 200 vértices da peça do cavalo criou um vértice extra, deslocou a numeração e fez a ferramenta gravar 9,53 MB, quando apenas 0,26 MB havia de fato mudado.

A solução foi construída em iterações, cada uma medida no Btrfs e registrada em um *commit* próprio, resumidas na Tabela 4. Das onze etapas, sete foram adotadas, uma serviu apenas para medição e três foram descartadas porque a medição mostrou que não traziam ganho.

Tabela 4 – Iterações de melhoria do gitbin e do *addon*

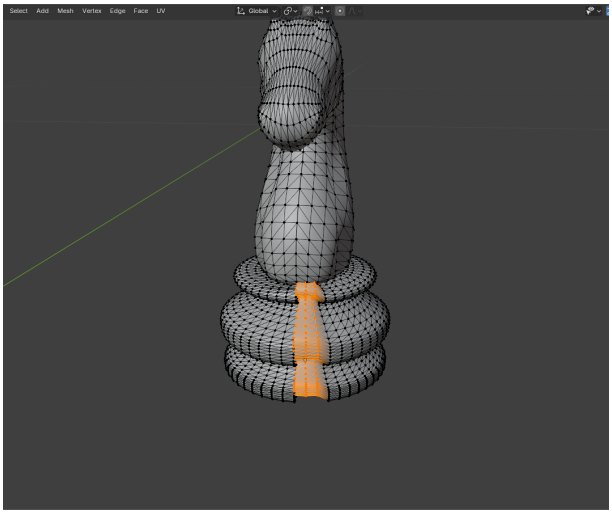
Iteração	Resultado observado	Situação
`init` sem privilégios de administrador	Imagem esparsa ocupa apenas o que foi gravado	Adotada
Medição do exportador padrão	Determinístico, preserva imagens, reconstrói a geometria	Medição
Opções do exportador (tangentes, *accessors*)	Nenhuma opção mantém a ordem dos vértices	Descartada
Comparação por conteúdo	Total da sessão de 89 para 47 MB	Adotada
Preservação dos *bytes* das malhas não editadas	Salvar sem editar passa a custar 0	Adotada
Ordem estável e gravação em blocos de 4 KiB	Base para gravar só a região alterada	Adotada
Vértices movidos regravados na ordem original	Mover 1 vértice foi de 4,59 para 0,01 MB	Adotada
Redução do custo fixo de metadados	De 96 para 24 KB por versão	Adotada
JSON do glTF separado por chave	Metadados extras superam a economia	Descartada
Índice herdado pela *bufferView* alterada	Ganho de 0,06%	Descartada
Impressão digital ignora atributos do modo de edição	Malha volta a ser preservada após edição	Adotada

Fonte: Autoria própria (2026).

O resultado foi uma divisão de responsabilidades coerente com as duas camadas do método, assim o Blender identifica o que mudou, a camada lógica monta o arquivo e a camada física guarda apenas a diferença. Ao importar o modelo, o *addon* marca cada malha com seu índice no glTF e calcula uma impressão digital, um *hash* SHA-256, como dito anteriormnte, de todos os seus atributos. Ao salvar, a cena é exportada normalmente e a impressão digital é recalculada. Nas malhas não editadas, o comando `gitbin mesclar` substitui a geometria

exportada pelos *bytes* originais e já nas malhas em que apenas vértices foram movidos, como no exemplo da Figura 10, gera as novas posições na ordem original, de modo que somente os blocos que contêm esses vértices são regravados. Em seguida, o `gitbin save` reconhece conteúdos já existentes, mesmo em outra posição, reescreve somente os blocos de 4 KiB alterados, e o *snapshot* do Btrfs compartilha fisicamente todos os demais.

Figura 10 – Vértices editados, em laranja, na peça do cavalo



Fonte: Autoria própria (2026).

Algumas limitações permanecem. Alterações de topologia, como criar ou apagar faces, ainda fazem a malha inteira ser regravada, pois ela passa a vir do exportador em outra ordem. Texturas editadas são recomprimidas em JPEG pelo Blender, o que altera a imagem inteira, característica inerente a formatos comprimidos.

4.4 Discussão

Esta seção discute os resultados dos experimentos finais, nos quais o gitbin e o *addon* do Blender foram comparados ao Git LFS em três modelos tridimensionais, o *ABeautifulGame*, 41 MB, usado nas etapas anteriores, o *Spring*[2], 339 MB, cujo tamanho é dominado por texturas, e o *San Miguel*[3], 344 MB, formado apenas por geometria. Em cada modelo foi executada a mesma sessão de dez versões no Blender, que vai do arquivo original a edições como mover vértices, alterar um material, adicionar e apagar objetos e apagar faces, e cada combinação de modelo e método foi executada em uma máquina virtual nova e idêntica. As figuras a seguir apresentam os resultados em ordem, do espaço ocupado ao custo em tempo e memória.

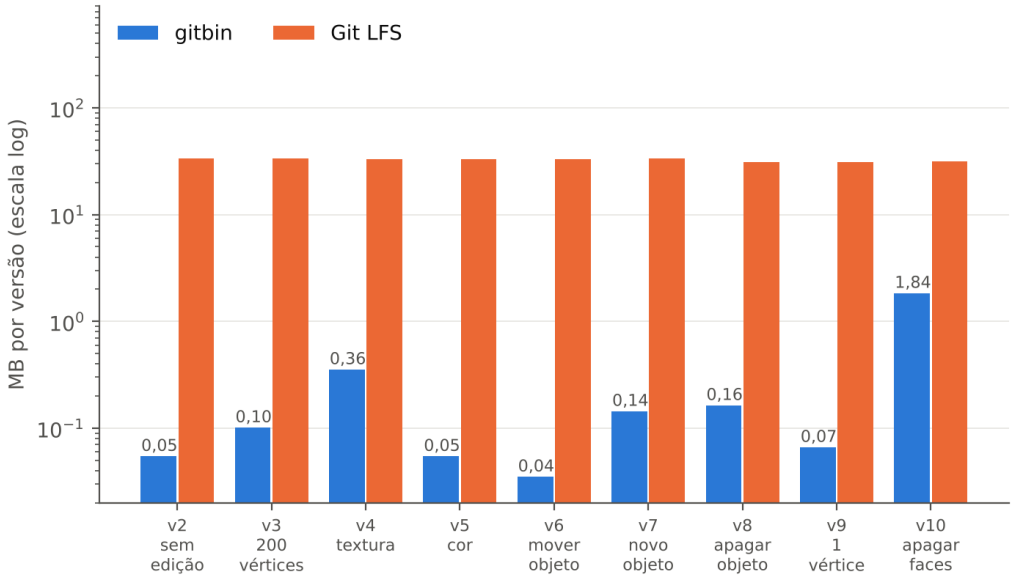
[2] BLENDER STUDIO. *Spring*: personagem. 2019. Disponível em: https://studio.blender.org/characters/spring/v1/. Acesso em: 11 nov. 2026.

[3] LEAL LLAGUNO, Guillermo M. *San Miguel 2.0*. In: MCGUIRE, Morgan. *Computer Graphics Archive*. 2017. Disponível em: https://casual-effects.com/data/. Acesso em: 11 nov. 2026.

4.4.1 Custo de cada versão

As Figuras 11, 12 e 13 mostram quanto o repositório cresceu a cada versão em cada método, no *ABeautifulGame*, no *Spring* e no *San Miguel*, respectivamente. Na primeira versão, os dois métodos guardam o arquivo inteiro e, por isso, têm custo semelhante. A partir da segunda, o comportamento se separa. O Git LFS cresce quase o tamanho do arquivo em todas as versões, cerca de 33 MB no *ABeautifulGame*, 360 MB no *Spring* e 387 MB no *San Miguel*, qualquer que seja a edição feita. Isso ocorre mesmo ao salvar sem editar, pois o Blender exporta um arquivo ligeiramente diferente do original e o Git LFS, ao tratá-lo como uma unidade indivisível, o guarda por completo.

Figura 11 – Crescimento do repositório a cada versão no *ABeautifulGame*

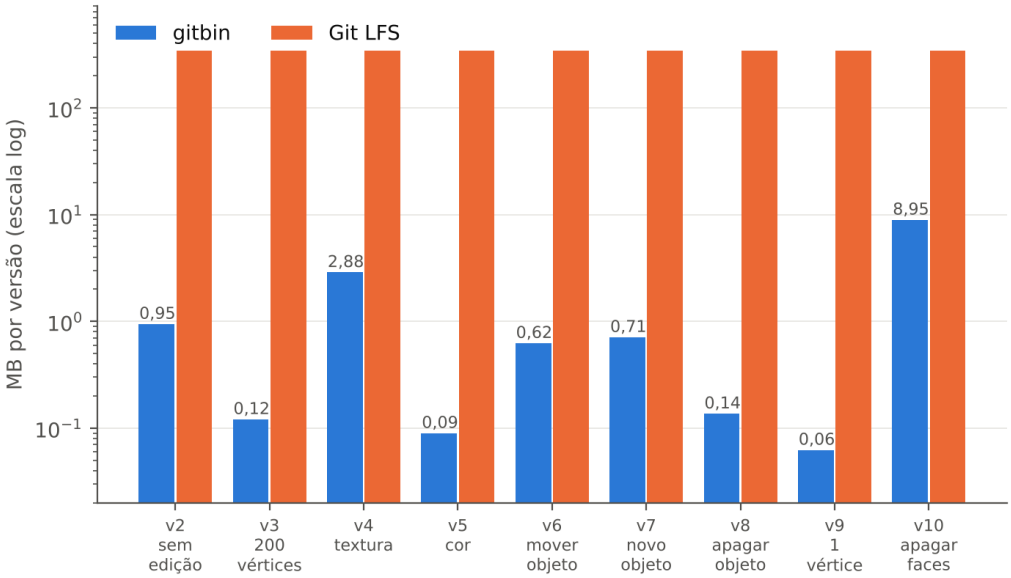


Fonte: Autoria própria (2026).

No gitbin, o custo acompanha o tamanho da alteração. Edições pequenas e localizadas, como mover vértices ou mudar a cor de um material, gravaram de 0,04 MB a 0,12 MB no *ABeautifulGame* e no *Spring*, menos de 0,5% do custo do Git LFS para a mesma edição. As edições que alteram a estrutura do modelo custam mais, pois editar uma textura regrava a imagem inteira e apagar faces, versão 10, regrava a malha inteira, que volta do exportador em outra ordem. Essa foi a edição mais cara em todos os modelos, com 1,84 MB, 8,95 MB e 14 MB, e ainda assim ficou abaixo de 6% do custo do Git LFS. No *San Miguel*, a versão 4 foi omitida, porque o modelo não possui texturas.

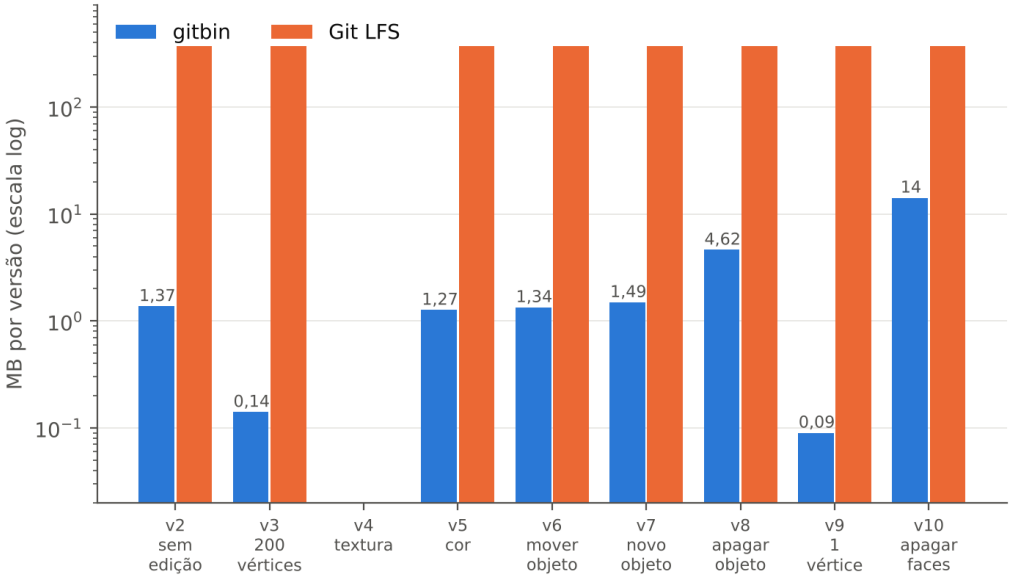
As Figuras 14, 15 e 16 mostram o que o gitbin fez com cada *bufferView* a cada versão, nos três modelos. Em todos os modelos, a maior parte delas permaneceu inalterada ou foi reaproveitada, e apenas uma pequena fração foi de fato gravada, ou seja, as *bufferViews* modificadas ou novas, que nunca passaram de cerca de 8% do total. Nas edições pequenas, como salvar sem editar, mover um vértice ou mudar a cor de um material, mais de 95% das *bufferViews* foram compartilhadas entre as versões, sem nenhuma gravação.

Figura 12 – Crescimento do repositório a cada versão no *Spring*



Fonte: Autoria própria (2026).

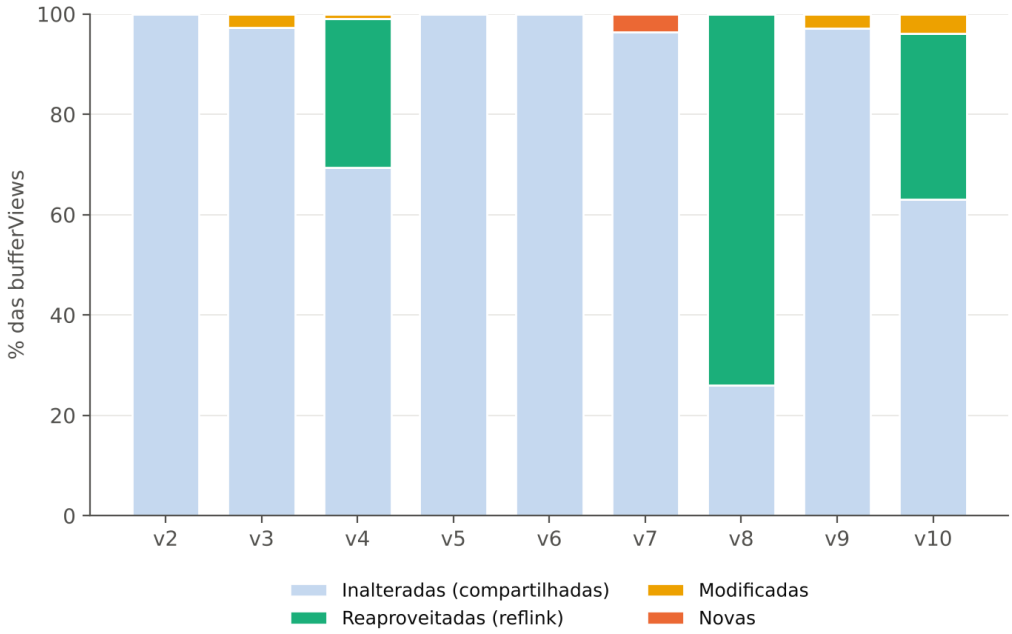
Figura 13 – Crescimento do repositório a cada versão no *San Miguel*



Fonte: Autoria própria (2026).

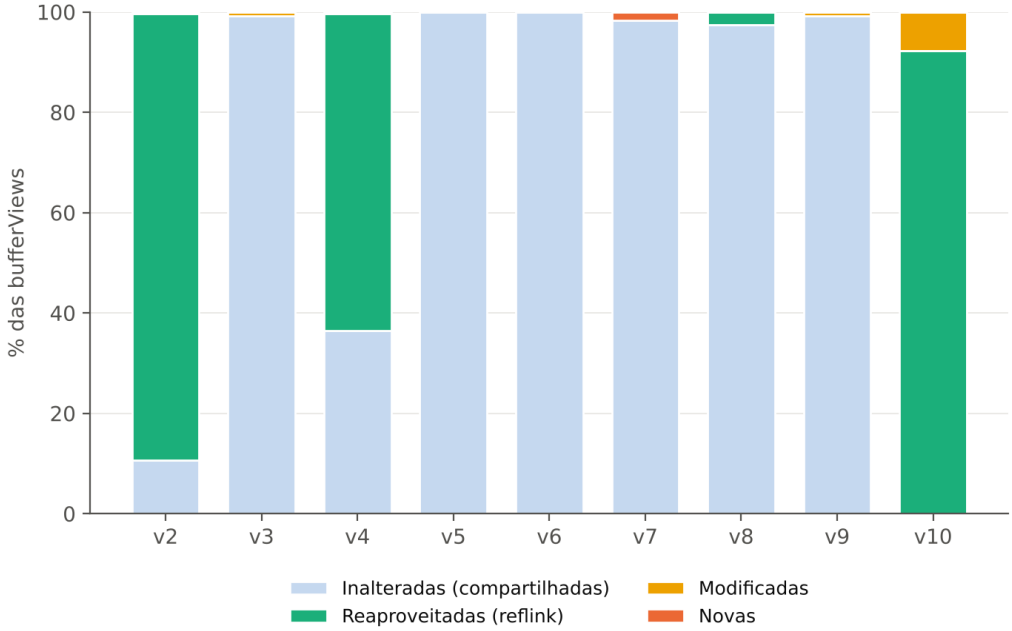
As *bufferViews* reaproveitadas são aquelas cujo conteúdo já existia na versão anterior, mas em outra posição do arquivo. Isso ocorre quando um objeto é apagado, pois as *bufferViews* seguintes mudam de índice. Nesse caso, o gitbin as reconhece pelo *hash* e as reaproveita por *reflink*, sem copiar os dados, o que aparece na versão 8, em que a metade ou mais delas foi reaproveitada. Esse mecanismo evita que o deslocamento de índices seja tratado como modificação, problema que motivou a comparação por conteúdo descrita na Seção 4.3.

Figura 14 – Classificação das *bufferViews* do arquivo a cada versão no *ABeautifulGame*



Fonte: Autoria própria (2026).

Figura 15 – Classificação das *bufferViews* do arquivo a cada versão no *Spring*

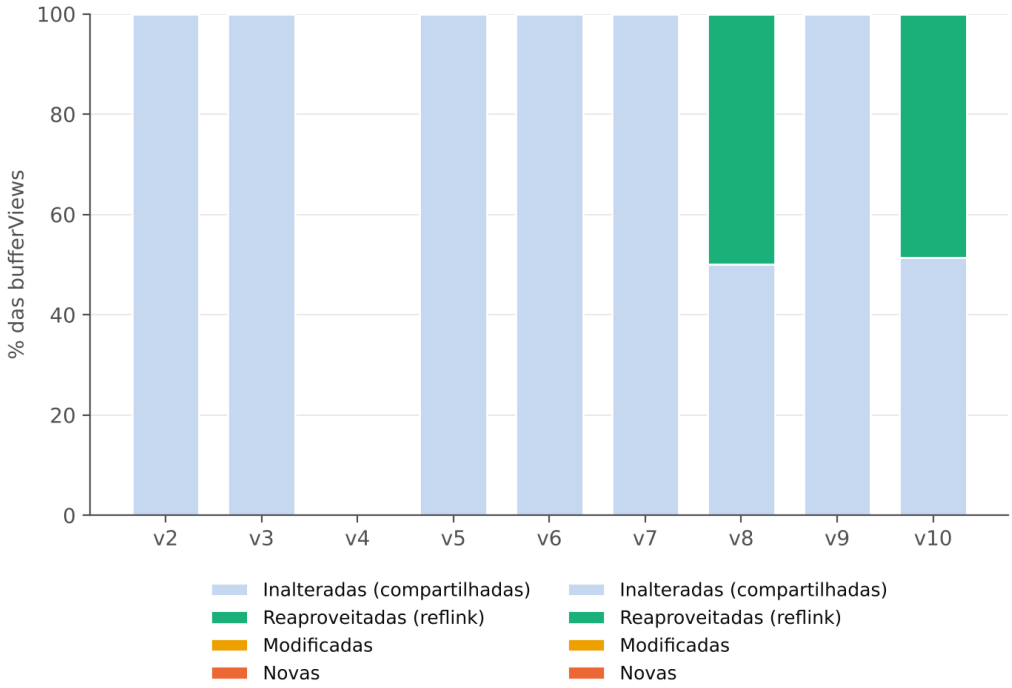


Fonte: Autoria própria (2026).

4.4.2 Custo em tempo e memória

A economia de espaço tem um custo em tempo de processamento. As Figuras 17, 18 e 19 mostram o tempo médio do *save* por etapa nas versões 2 a 10, em cada modelo. O gitbin levou 2,5 s, 22,5 s e 33,3 s nos três modelos, contra 1,1 s, 14,0 s e 13,5 s do Git LFS. A maior parte da diferença vem da exportação feita pelo *addon*, que calcula a impressão digital de

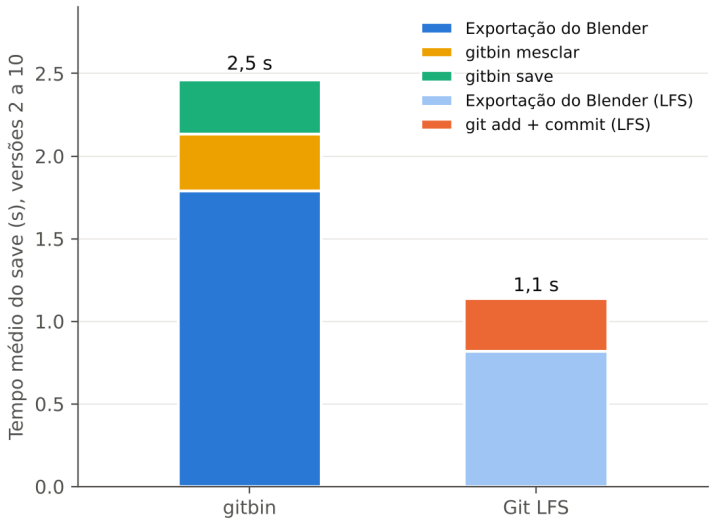
Figura 16 – Classificação das *bufferViews* do arquivo a cada versão no *San Miguel*



Fonte: Autoria própria (2026).

cada malha, e não da camada física, pois os comandos `gitbin mesclar` e `gitbin save` somados ocuparam uma parcela menor do tempo. O efeito cresce com o número de malhas e é mais visível no *San Miguel*, com 1004 malhas.

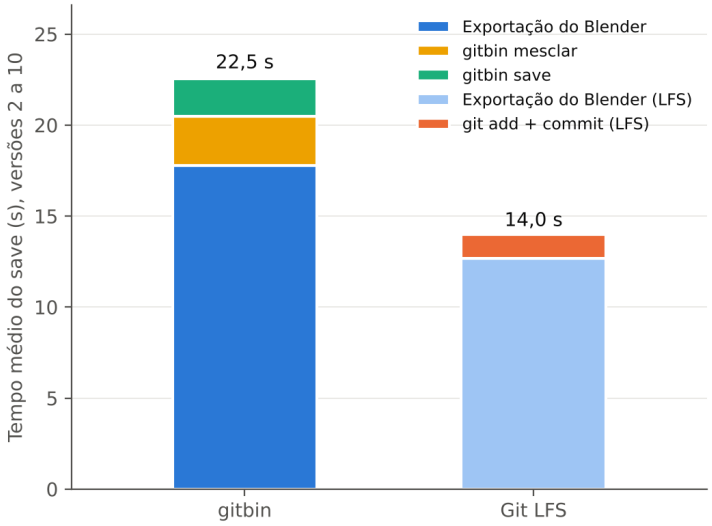
Figura 17 – Tempo médio do *save* por etapa no *ABeautifulGame*



Fonte: Autoria própria (2026).

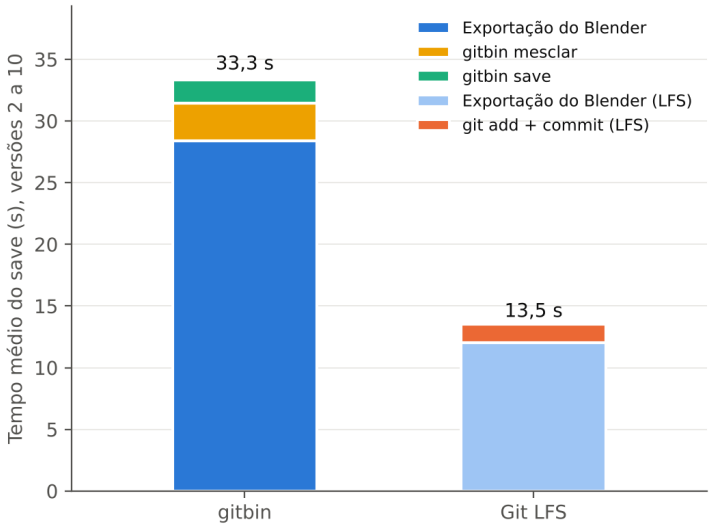
As Figuras 20, 21 e 22 resumem o custo do gitbin em relação ao Git LFS no tempo do *save*, no pico de memória e no tempo de restauração, respectivamente. O *save* foi de 1,6 a 2,5 vezes mais lento, e o pico de memória do Blender foi de 1,1 a 1,3 vez maior, ainda dentro do que uma máquina com 16 GB comporta. A restauração de uma versão foi a operação mais afetada,

Figura 18 – Tempo médio do *save* por etapa no *Spring*



Fonte: Autoria própria (2026).

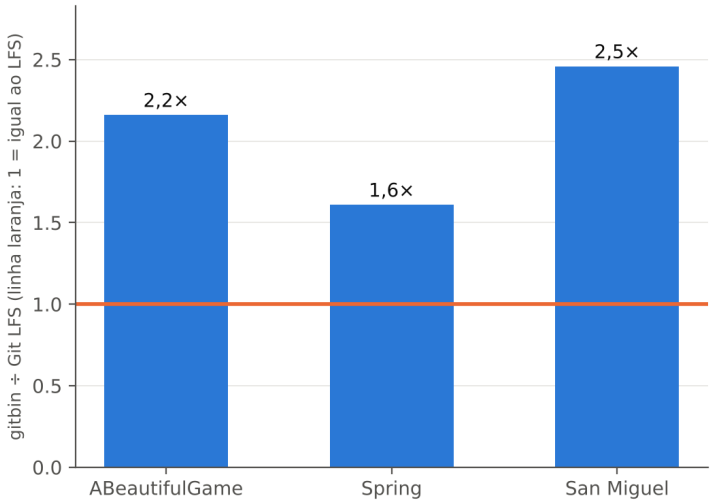
Figura 19 – Tempo médio do *save* por etapa no *San Miguel*



Fonte: Autoria própria (2026).

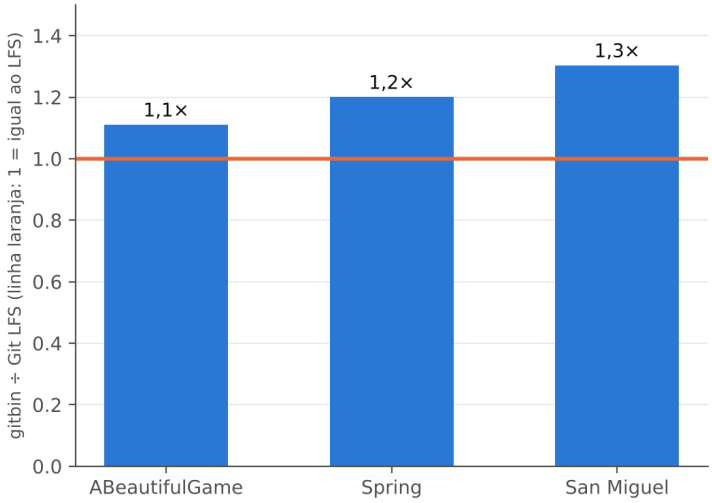
de 4,9 a 20,1 vezes mais lenta, pois o gitbin precisa remontar o arquivo a partir de milhares de segmentos, enquanto o Git LFS apenas copia um arquivo. No *San Miguel*, isso equivaleu a 8,4 s contra 0,4 s. O método troca, portanto, tempo e memória por espaço em disco, o que tende a ser aceitável quando as versões são salvas poucas vezes por hora e restauradas raramente.

Figura 20 – Razão de tempo do *save*, gitbin em relação ao Git LFS



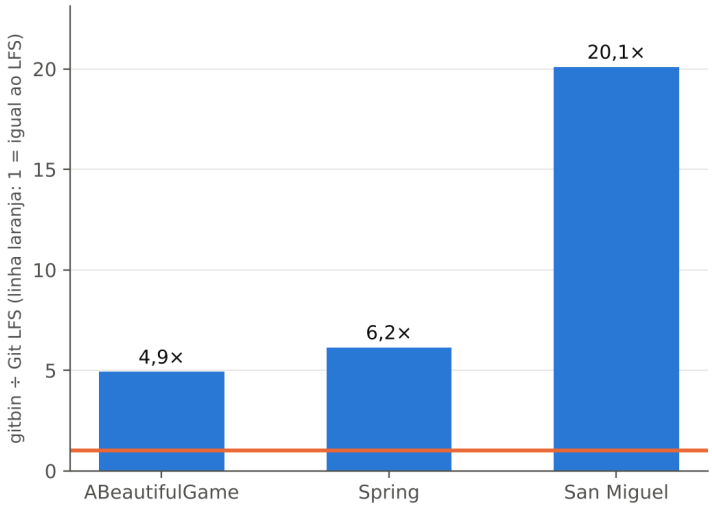
Fonte: Autoria própria (2026).

Figura 21 – Razão do pico de memória do Blender, gitbin em relação ao Git LFS



Fonte: Autoria própria (2026).

Figura 22 – Razão do tempo de restauração, gitbin em relação ao Git LFS



Fonte: Autoria própria (2026).

5 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, validou a viabilidade da abordagem. Em três modelos tridimensionais, *ABeautifulGame*, *Spring* e *San Miguel*, submetidos a uma sessão de dez versões de edição no Blender, o método proposto reduziu o espaço ocupado em 89,2%, em média. No *San Miguel*, por exemplo, o acumulado foi de 377,59 MB contra 3294,02 MB do Git LFS. O custo de cada versão no gitbin acompanhou o tamanho da alteração, e edições pequenas e localizadas gravaram de 0,04 MB a 0,12 MB no *ABeautifulGame* e no *Spring*, enquanto o Git LFS gravou quase o tamanho do arquivo a cada versão. Em todas as versões medidas, o arquivo restaurado teve conteúdo idêntico ao original e passou pelo validador do glTF sem erros.

A economia de espaço teve como contrapartida um custo em tempo e memória. O *save* foi de 1,6 a 2,5 vezes mais lento que o do Git LFS, o pico de memória do Blender foi de 1,1 a

1,3 vez maior e a restauração de uma versão foi de 4,9 a 20,1 vezes mais lenta. Em valores absolutos, esse custo ficou na casa de segundos, e a economia de espaço, por sua vez, se acumula a cada versão salva. O método se mostra, assim, vantajoso quando o espaço em disco pesa mais que o tempo de cada operação, como em repositórios de modelos grandes e com muitas versões. Por terem sido obtidos com três modelos, no Blender e com uma sessão de edição por modelo, esses resultados indicam uma tendência que pode ser confirmada com outros modelos e com o uso por várias pessoas ao mesmo tempo.

Os resultados obtidos permitem especular aplicações concretas do método. Na indústria de jogos digitais, estúdios que mantêm repositórios com centenas de modelos GLB modificados diariamente por equipes de artistas se beneficiariam ao ver edições pequenas gravarem de 0,04 MB a 0,12 MB, em vez de dezenas ou centenas de megabytes por versão.

Por fim, o método proposto não busca se restringir ao formato GLB, qualquer formato binário com estrutura interna segmentada e *offsets* explícitos apresenta as condições necessárias para sua aplicação. Estudos futuros podem avaliar sua extensão a outros formatos amplamente utilizados na indústria, como o *Volumetric Dynamic B plus tree* (VDB) inicialmente, formato volumétrico adotado em simulações de fluidos e efeitos visuais. Experimentos comparativos entre volumes Btrfs e ext4 permitiriam isolar o impacto do sistema de arquivos das demais variáveis do método, e a avaliação do uso por várias pessoas ao mesmo tempo permitiria verificar o método em fluxos de trabalho colaborativos.

REFERÊNCIAS

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

BENITES, V. d. F.; FLORES, G. L.; RODRIGUES, F. B. **Basic Solution File System**. Campo Grande, 2025.

CHEN, H.-T.; WEI, L.-Y.; CHANG, C.-F. Nonlinear revision control for images. **ACM Transactions on Graphics**, New York, v. 30, n. 4, p. 1–10, jul. 2011. Article 105.

CHEN, J. *et al.* 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, Y. *et al.* Unveiling security vulnerabilities in Git Large File Storage protocol. *In*: **IEEE Symposium on Security and Privacy (SP)**. [*S.l.*]: IEEE, 2025.

CUNHA, M. B. **Entendendo o uso do Git em equipes de desenvolvimento de software**. 2018. Dissertação (Trabalho de Conclusão de Curso) — Universidade Federal de Pernambuco, Recife, 2018.

DOBOŠ, J.; STEED, A. 3D revision control framework. *In*: **International Conference on 3D Web Technology (Web3D)**. New York: [*s.n.*], 2012. Los Angeles.

HOUSTON, B. **glTF**: everything you need to know. 2019. Threekit Blog. 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.*], 1991. 3-22–3-31 p. Acesso em: 09 maio 2026.

KASAMPALIS, S. **Copy On Write Based File Systems Performance Analysis And Implementation**. 2010. 83 p. Dissertação (Mestrado) — Department of Informatics, Technical University of Denmark, Kongens Lyngby, 2010.

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

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

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

MACDONALD, J. P. **File System Support for Delta Compression**. 2000. Dissertação (Mestrado) — University of California at Berkeley, Berkeley, 2000.

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

RODEH, O. B-trees, shadowing, and clones. *In*: **Linux Storage and File System Workshop**. [*s.n.*], 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, D. 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, W. *et al.* Design and analysis of block-level snapshots for data protection and recovery. **IEEE Transactions on Computers**, IEEE Computer Society, v. 58, 2009.

ZHENG, S. *et al.* HMVFS: a hybrid memory versioning file system. *In*: **32nd Symposium on Mass Storage Systems and Technologies (MSST)**. Santa Clara, CA: IEEE, 2016. p. 1–12.

APÊNDICE A – Dados complementares dos experimentos finais

Este apêndice reúne os dados completos dos experimentos finais descritos no Capítulo 4, nos quais o gitbin e o Git LFS foram comparados em três modelos tridimensionais. Os valores de espaço, escrita em disco e memória estão em megabytes (MB) e gigabytes (GB) binários, e os tempos estão em segundos. Nas tabelas, a versão 4 do *San Miguel* aparece com traços, pois essa edição foi omitida porque o modelo não possui texturas.

A.1 Ambiente e modelos

A Tabela 5 apresenta a configuração das máquinas virtuais, idêntica nas seis rodadas, e a Tabela 6 descreve os três modelos utilizados.

Tabela 5 – Configuração das máquinas virtuais

Item	Configuração
Virtualização	Multipass (QEMU/KVM), uma máquina nova por rodada
Sistema operacional	Ubuntu 26.04.1 LTS, *kernel* 7.0.0-38
Recursos	4 *vCPUs*, 16 GB de memória e 30 GB de disco (esparso)
Hardware do hospedeiro	Intel Core i5-10400F, 2,90 GHz
Blender e Python	Blender 5.2.2 LTS e Python 3.14.4
Versionamento	Git 2.53.0; Git LFS 3.7.1 (rodadas do Git LFS); `btrfs-progs` 6.17.1 (rodadas do gitbin)
Medição	sysstat 12.7.7 (`iostat`) e GNU `time`
Validação	Khronos glTF-Validator 2.0.0-dev.3.10

Fonte: Autoria própria (2026).

Tabela 6 – Características dos modelos utilizados

Característica	*ABeautifulGame*	*Spring*	*San Miguel*
Tamanho do arquivo (MB)	41,0	339,3	343,9
Malhas	15	27	1004
Nós	49	2288	1004
BufferViews	108	236	5980
Materiais	15	18	287
Imagens	33	24	0
Vértices	417.028	1.026.094	9.326.145
Triângulos	573.952	1.589.280	9.963.191
Ossos do *rig*	0	2260	0

Fonte: Autoria própria (2026).

A.2 Resultados por versão

As tabelas a seguir apresentam, para cada versão, o crescimento do repositório, o tempo do *save*, o pico de memória do Blender, o tempo de restauração e a escrita em disco medida com o `iostat`. As versões correspondem às edições da sessão descrita na Seção 4.4.

Tabela 7 – Crescimento do repositório a cada versão (MB)

		ABeautifulGame		*Spring*		*San Miguel*	
v	**Edição**	gitbin	LFS	gitbin	LFS	gitbin	LFS
1	Original	33,08	41,02	339,92	339,32	353,11	343,90
2	Salvar sem editar	0,05	33,36	0,95	343,38	1,37	368,75
3	Mover 200 vértices	0,10	33,36	0,12	343,39	0,14	368,75
4	Editar textura	0,36	33,30	2,88	343,91	—	—
5	Cor de material	0,05	33,30	0,09	343,91	1,27	368,75
6	Mover objeto	0,04	33,30	0,62	343,91	1,34	368,75
7	Adicionar objeto	0,14	33,37	0,71	343,97	1,49	368,81
8	Apagar objeto	0,16	31,12	0,14	343,23	4,62	368,77
9	Mover 1 vértice	0,07	31,12	0,06	343,23	0,09	368,77
10	Apagar 10 faces	1,84	31,55	8,95	343,23	14,16	368,77
Total acumulado		35,90	334,82	354,44	3431,47	377,59	3294,02

Fonte: Autoria própria (2026).

Tabela 8 – Tempo do *save* a cada versão (s)

		ABeautifulGame		*Spring*		*San Miguel*	
v	**Edição**	gitbin	LFS	gitbin	LFS	gitbin	LFS
1	Original	0,33	0,20	2,24	1,30	7,70	1,40
2	Salvar sem editar	2,64	1,34	23,95	13,36	33,37	14,55
3	Mover 200 vértices	2,53	1,08	21,32	12,95	34,34	13,20
4	Editar textura	2,40	1,12	22,28	14,13	—	—
5	Cor de material	2,28	1,10	24,67	15,46	31,78	13,42
6	Mover objeto	2,29	1,23	22,17	14,12	31,89	13,24
7	Adicionar objeto	2,31	0,99	21,63	13,90	31,83	13,12
8	Apagar objeto	2,33	1,16	21,84	13,97	35,48	13,75
9	Mover 1 vértice	2,52	1,09	22,62	14,11	32,88	13,08
10	Apagar 10 faces	2,83	1,12	22,29	13,92	34,87	13,96
Média, versões 2 a 10		2,46	1,14	22,53	13,99	33,30	13,54

Fonte: Autoria própria (2026).

A.3 Integridade das versões

A Tabela 12 resume a verificação feita ao final de cada rodada, na qual cada versão foi restaurada, teve seu conteúdo comparado ao que havia sido salvo e foi validada com o validador do Khronos Group.

Tabela 9 – Pico de memória do Blender a cada versão (GB)

		ABeautifulGame		*Spring*		*San Miguel*	
v	**Edição**	gitbin	LFS	gitbin	LFS	gitbin	LFS
1	Original	0,28	0,30	0,28	0,30	0,28	0,30
2	Salvar sem editar	1,23	1,18	4,25	3,48	4,89	3,65
3	Mover 200 vértices	1,35	1,25	4,51	3,74	5,16	3,98
4	Editar textura	1,49	1,30	4,70	3,85	—	—
5	Cor de material	1,51	1,33	4,66	3,86	5,30	4,07
6	Mover objeto	1,57	1,34	4,32	3,88	4,95	4,36
7	Adicionar objeto	1,55	1,34	4,66	3,81	5,33	4,01
8	Apagar objeto	1,51	1,37	4,66	3,89	5,39	4,12
9	Mover 1 vértice	1,40	1,36	4,73	3,89	5,42	4,11
10	Apagar 10 faces	1,52	1,36	4,77	3,95	5,90	4,16
Máximo		1,57	1,37	4,77	3,95	5,90	4,36

Fonte: Autoria própria (2026).

Tabela 10 – Tempo de restauração (*checkout*) a cada versão (s)

		ABeautifulGame		*Spring*		*San Miguel*	
v	**Edição**	gitbin	LFS	gitbin	LFS	gitbin	LFS
1	Original	0,37	0,08	2,04	0,34	8,00	0,37
2	Salvar sem editar	0,35	0,08	2,05	0,36	11,53	0,48
3	Mover 200 vértices	0,35	0,07	2,43	0,35	8,31	0,46
4	Editar textura	0,36	0,07	2,09	0,35	—	—
5	Cor de material	0,35	0,07	2,05	0,35	7,92	0,39
6	Mover objeto	0,37	0,07	2,13	0,35	7,92	0,37
7	Adicionar objeto	0,36	0,08	2,29	0,36	8,09	0,43
8	Apagar objeto	0,34	0,07	2,12	0,35	8,16	0,41
9	Mover 1 vértice	0,37	0,07	2,04	0,35	7,76	0,42
10	Apagar 10 faces	0,36	0,07	2,28	0,34	8,17	0,43
Média, versões 2 a 10		0,36	0,07	2,16	0,35	8,48	0,42

Fonte: Autoria própria (2026).

Tabela 11 – Dados gravados em disco a cada versão, medidos com o `iostat` (MB)

		ABeautifulGame		*Spring*		*San Miguel*	
v	**Edição**	gitbin	LFS	gitbin	LFS	gitbin	LFS
1	Original	33,2	41,3	340,0	339,6	353,2	344,2
2	Salvar sem editar	0,2	69,4	1,1	689,7	1,7	740,5
3	Mover 200 vértices	0,2	67,0	0,3	687,1	0,5	737,9
4	Editar textura	0,5	66,9	3,1	688,1	—	—
5	Cor de material	0,2	66,9	0,2	688,1	1,6	737,9
6	Mover objeto	0,2	66,9	0,8	688,2	1,7	737,9
7	Adicionar objeto	0,3	67,0	0,9	688,3	1,9	738,0
8	Apagar objeto	0,3	62,5	0,3	686,9	5,5	738,0
9	Mover 1 vértice	0,2	62,5	0,2	686,8	0,5	738,0
10	Apagar 10 faces	2,0	63,4	9,2	686,9	15,2	738,0
Total		37,1	633,7	356,1	6529,7	381,6	6250,6

Fonte: Autoria própria (2026).

Tabela 12 – Resultado da verificação das versões restauradas

Modelo	**Método**	**Versões**	**Conteúdo idêntico**	**Erros**	**Avisos (mín.–máx.)**
ABeautifulGame	gitbin	10	10 de 10	0	0–1
ABeautifulGame	Git LFS	10	10 de 10	0	0–15
Spring	gitbin	10	10 de 10	0	20–37
Spring	Git LFS	10	10 de 10	0	20–37
San Miguel	gitbin	9	9 de 9	0	0–0
San Miguel	Git LFS	9	9 de 9	0	0–0

Fonte: Autoria própria (2026).