Tampere University

Joel Alanko

DYNAMIC BENCHMARK FOR GRAPHICS RENDERING

Master's Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Dr. Markku Mäkitalo
Prof. Pekka Jääskeläinen
April 2021

ABSTRACT

Joel Alanko: Dynamic benchmark for graphics rendering
Master's Thesis
Tampere University
Degree Programme in Information Technology, MSc (Tech)
April 2021

Most graphics rendering algorithms used in both animated feature films and real time games can enjoy the performance and quality boost that comes with temporally reusing previous computation. However, there is a lack of proper rendering benchmarks that would allow people to have detailed and objective comparisons between different temporal methods. Currently, very slowly moving cameras, improper scenes, and animations are used, which results in an unequaled playground for comparisons, having an obvious bias towards the proposed novel methods.

In this thesis, we describe a framework that can be used to capture 3D animations out of interactive scenarios and compile them to a dataset that is compatible as a dynamic benchmark. The capturing framework is used in the creation of two datasets: EternalValleyVR and EternalValleyFPS. We verify the quality and the dynamic challenge these datasets put on the algorithms. By surveying the input features used in the state of the art temporal reuse algorithms, we form metrics of change in features that happen throughout the animation. The proposed dynamic benchmarks are shown to surpass the previously released animations in temporal complexity.

Keywords: Graphics, Rendering, Temporal Reuse, Path Tracing, Ray Tracing, Dataset, Benchmark

TIIVISTELMÄ

Joel Alanko: Dynaaminen suorituskykytesti grafiikan renderöintiin
Tampereen yliopisto
Tietotekniikan DI-tutkinto-ohjelma
Huhtikuu 2021

Useimpia renderöintialgoritmeja, joita käytetään elokuvissa ja peleissä, voidaan nopeuttaa ja parantaa laadullisesti uudelleenkäyttämällä aiemmin laskettua informaatiota. Mutta näille temporaalisille algoritmeille ei ole olemassa kunnollisia vertailutestejä. Tällä hetkellä renderöintialgoritmeja testataan hitailla kameroilla, kertakäyttöanimaatioilla sekä testiskeneillä. Tämä johtaa selvästi epäreiluun asetelmaan verrattavien algoritmien kanssa, koska skenet voivat olla suunniteltu näyttämään omat algoritmit hyvässä valossa.

Tässä diplomityössä luodaan tallennusjärjestelmä, jolla voi taltioida temporaalisiin vertailutesteihin sopivia dynaamisia 3D-animaatioita. Järjestelmää demonstroidaan luomalla kaksi vertailutestitiedostoa EternalValleyVR ja EternalValleyFPS. Luotujen animaatioiden temporaalinen laatu halutaan varmistaa. Käytännössä tämä tehdään perehtymällä moderneihin temporaalisiin algoritmeihin, sekä luomalla niiden käyttämien parametrien perusteella sopivat vertailumetriikat. Metriikoita käytetään kahden luodun testitiedoston vertaamiseen yleisesti käytössä oleviin animaatioihin. Vertailun perusteella EternalValleyVR ja EternalValleyFPS ovat huomattavasti hankalampia renderöitäviä aikaisemmin julkaistuihin animaatioihin verrattuna.

Avainsanat: Grafiikka renderöinti, Temporaalinen uusiokäyttö, Polunseuranta, Säteenseuranta, Datasetti, Suorituskykytesti

PREFACE

This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 783162 (FitOptiVis). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Netherlands, Czech Republic, Finland, Spain, Italy.

Work was done at Tampere University while working as a research assistant in the university's Virtual Reality and Graphics group. First and foremost, I would like to thank Prof. Pekka Jääskeläinen for the opportunity to finish my thesis and trust to work on this subject, and my supervisor Dr. Markku Mäkitalo, for the guidance and feedback.

I would like to thank Julius for suggesting the initial idea and inspiration for this thesis. Thanks to the rest of the graphics group for the fruitful and informative brainstorming. Indeed, the work would be of less if not for those conversations.

In addition, I'd like to thank my family and friends, who have, throughout my years of study, given me tireless love and support. Especially I'd like to thank my grandfather Risto for the compassionate encouragements in my studies. Finally, I want to thank Riina for relentlessly cheering me up while writing the thesis.

Tampere, 26th April 2021

Joel Alanko

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

B	Local bone transformation matrix
L_e	Power of radiance emitted from ray hit surface
L_{in}	Incoming radiance to a point
L_{out}	Radiance leaving a point to a direction
M	Bind pose matrix
R	Rotation matrix used in transformations
S	Scale matrix used in transformation
T	Translation vector used in transformations
Ω	Hemisphere over a ray hit point
$\alpha_{discard}$	Threshold value for the discard function
α	Temporal accumulation blending factor
δ	Angle of pitch, yaw or roll rotation
ϕ	Angle of rotation around a vector
$\pi(\cdot)$	Reprojection operator
θ	Angle between surface normal and incoming light direction
c	Camera position vector
d_p	Change in translation
d_r	Change in rotation angle
d	Dual quaternion
f_n	Function to accumulate pixel's color history
f_r	Bidirectional reflectance distribution function
$f_{discard}$	Discard function for invalid pixels
$f_{percentage}$	Function to retrieve the percentage of image pixels discarded
p, n, u, v, c	Vector-valued variables
q	Unit quaternion
t	Time parameter
x, y, z, w, h	Single value variables

3D	Three dimensional
AI	Artificial intelligent
API	Application programming interface
APSNR	Average of peak to signal noise ratio in animation
ASCII	American standard code for information interchange is a character encoding
BART	A benchmark for animated ray tracing
BRDF	Bidirectional reflectance distribution function
CAD	Computer aided design
CBR	Checkerboxing rendering method
CC-0 licence	Creative commons no rights reserved licence
CPU	Graphics processing unit
DAE	Collada is 3D file format by Khronos Group
DLSS	Deep learning supersampling
FBX	Filmbox is a proprietary file format by Autodesk
GGX	A microfacet distribution
glTF 2.0	Graphics language transform format version 2.0 specification
GPU	Graphics programming unit
HMD	Head mounted display
HUD	Heads up display
IP	Intellectual property
KAIST	3D Model benchmark
MD5	3D file format by id Software
MPI	Max Planck Institute
MSAA	Multisample anti-aliasing
MTL	File format for material definition
NVidia RTX	NVidia ray tracing hardware
OBJ	Fileformat for geometry definition by Wavefront Technologies
ORCA	NVidia open research content archive
PBR	Physically based rendering, a modern material model
PBRT	Physically based research renderer
PC	Personal computer

PSNR	Peak to signal noise ratio
RGB	Additive color model with red, green and blue lights
spp	Samples per pixel
TAA	Temporal anti-aliasing
TRS	Affine transformation matrix
UNC	University of North Carolina
USD	Universal scene description is 3D file format from Pixar
UT AnimRep	The utah 3D animation repository
VR	Virtual reality

1 INTRODUCTION

Graphics rendering refers to methods for synthesizing from a virtual three-dimensional mathematical model to a display. Graphics rendering is commonplace in entertainment industries like animated feature films and games. It is often utilized in design phases with *computer aided design* (CAD) across almost all other industries, including industrial, product, and architecture industries.

A pleasant and interactive experience requires the display to update a new image in high frequency. However, rendering a realistic image takes time. It has been noticed that the next frame is often very coherent with the previous one in dynamic rendering. This coherency can be utilized so that the computational efforts are not wasted. These are called *temporal reuse* methods, which means that the previously rendered image is used in some way to help to render the following image.

Often when the performance of methods and processes is compared, *benchmarks* are created and used. Benchmarks contain reproducible test scenarios that are used as an input for algorithms. The results can then be compared with the confidence that the test was performed in an appropriate setting.

For temporal reuse algorithms, a benchmarking setting would be a dataset that contains 3D data and animations required in the image rendering. With benchmarks, it would be easier to compare the advancements in algorithm development, having access to previously understood and used dynamic datasets. Moreover, a benchmark would clarify the field of temporal rendering, showing how and where the state of the art algorithms succeed and fail in rendering good quality animations. It would also serve as a challenge to motivate pushing forward rendering development.

Dynamic datasets could also be helpful in the machine learning and deep learning neural network research that has recently gained much interest. The data-driven area of deep learning must have a large amount of data available so that the networks can learn as much as possible from the different inputs. In graphics rendering, this could be utilizing the quickly moving camera's transformation information in predictions of what the rendering should focus on. Also, the scenery deformations could determine what parts contribute to the final image. With such information, sophisticated occlusion culling or acceleration structure updates could be automated.

However, there are very few such animations released in public, and graphics research rarely uses them. There are a few obvious reasons for this. First, gathering and creating these datasets takes time and effort, and polishing them to a release quality would increase it even more [1]. Second, there is no single clear animation format to select from because there are plenty of standard file formats used across the industry. The papers use animations to produce convincing results, but rarely are publicly available datasets, or the used animations are released to the public. And three, the datasets are either created by the authors themselves, they own an IP they can use, or they buy an animation, which cannot be released to the public. The fact that animations are only present in the research papers' results serves as a bias towards the apparent novelties researchers are proposing, as it is impossible to reproduce the exact same case.

For comparison, there are standardized benchmarking datasets like these for static single image rendering algorithms. For example, the Sponza scene, with its simple geometry and reasonably complicated material models, is commonly used in real-time rendering algorithms, and the San Miguel scene is used with offline path tracing rendering algorithms [2]. Few datasets have animations, but they lack the most complicated scenarios that are commonplace in practice. Temporal rendering complexity comes with a quickly moving camera and fast-paced animations. Currently, these datasets only have slowly flying cameras interpolating from point to point, which does not often map with the actual use case [3]. The cameras shake and move irregularly when used in interactive scenarios.

In *virtual reality* (VR), the screen may shake even more than with PC or mobile applications, and it is also more sensitive to issues regarding bad quality rendering [4]. The VR research community is also lacking such dataset, and so do the other *head mounted display* (HMD), and screen rendering research areas of light-field and *augmented reality* (AR).

The signal processing research community uses a similar subset of temporal feature buffers in the motion flow algorithms [5, 6, 7]. Motion estimation can be used, for example, in automated car driving tasks. These datasets lack the required 3D world information for temporal reuse algorithms, but better datasets would help them too to generate new datasets for motion flow.

This thesis focuses on the following research questions:

- What are the fundamental features currently used in temporal reuse algorithms?
- What makes a temporally challenging dataset to be used in benchmarking?
- How can the temporal challenge be measured and compared between different animations?
- What are the existing dynamic benchmarks?

- How can the dataset be captured from an interactive scenario, like a game, and what is stored in the dataset?

This thesis's primary goal is to produce a framework that can be utilized to create dynamic datasets. We recognize that games already have complex animations accompanied by varying lighting settings that are hard to render in real-time, making an excellent dataset capture platform. To understand what makes an animation temporally challenging to render, we review graphics and 3D animation background for both real-time rasterization and path tracing and then familiarize ourselves with the modern temporal reuse algorithms. We analyze the inputs they often use and present metrics that can be used to compare datasets between each other. Finally, we use the proposed framework in the creation of two datasets and verify that the produced datasets have similar complexity in materials and geometry but show an increase in the challenge for the temporal algorithms.

The structure of the thesis is as follows. In the Chapter 2, we start by establishing the necessary 3D graphics rendering theory. Then, in the Chapter 3, we introduce the state of the art advancements in temporal reuse algorithms. We also present the benchmarking metrics that can be used to compare datasets' feature buffers we recognize utilized in temporal rendering algorithms. After that, in Chapter 4, we review previously released rendering benchmarks and dynamic datasets. Next, we present the capturing framework in the Chapter 5 and use it to capture and create two benchmarking datasets from an open-source game called Cube 2: Sauerbraten. Summarized results are presented in Chapter 6. Finally, in the conclusion Chapter 7, we discuss the work done and the outcome of the thesis.

2 GRAPHICS RENDERING

This Chapter describes necessary theory and methods standardized in computer graphics in creating and displaying virtual worlds to a computer screen. We start with declaring the rendering equation: simple concept, yet effective in practice giving realism to the 3D scenes by tying together computer graphics and actual physical lighting phenomena. The equation itself is approachable, but evaluating it has been the centermost issue in graphics rendering since it was introduced. Then we declare how surfaces react to the lighting using surface material definitions and their underlying microfacet structures. After that, we continue to show how geometry is produced out of simple mathematical concepts and primitives. Then, we show the standard practice on how virtual cameras and lights are described in computer graphics. Finally, we introduce rigidbody and armature animations and describe practical representations for moving and rotating objects in virtual worlds.

2.1 Light Transport

To synthesize an image to a screen, a standard way is to have a virtual camera that looks at 3D scene [8]. There are two popular approaches to this. First, the *rasterization* uses *graphics processing units* (GPU) to preprocess 3D world geometry primitives and finally calculate colors for each pixel in a vastly parallel manner. Rasterization is the standard for games, utilizing GPU *application programming interfaces* (API), like OpenGL, Vulkan and Direct3D12, to access function pointers and memory of the GPU [9, 10, 11]. The second common approach is *Monte Carlo path tracing*, which we will describe shortly. In both methods, conceptually, each pixel is colored according to a ray shot from a virtual camera hitting a point in the scene, using the underlying material of the hit surface. Like in real life, the lighting condition matters. The apparent color of, for example, a red apple is different when looked outdoors and when observed in a dark cellar.

The rendering equation is an integral equation for all the *radiance* leaving a point to a direction. Radiance is the energy of a light source. The equation was introduced simultaneously by Kajiya and Immel et.al. [12, 13]. Simply, the equation is a sum of reflected radiance and emission:

$$L_{out} = L_e + \int_{\Omega} L_{in} f_r cos\theta d\omega, \tag{2.1}$$

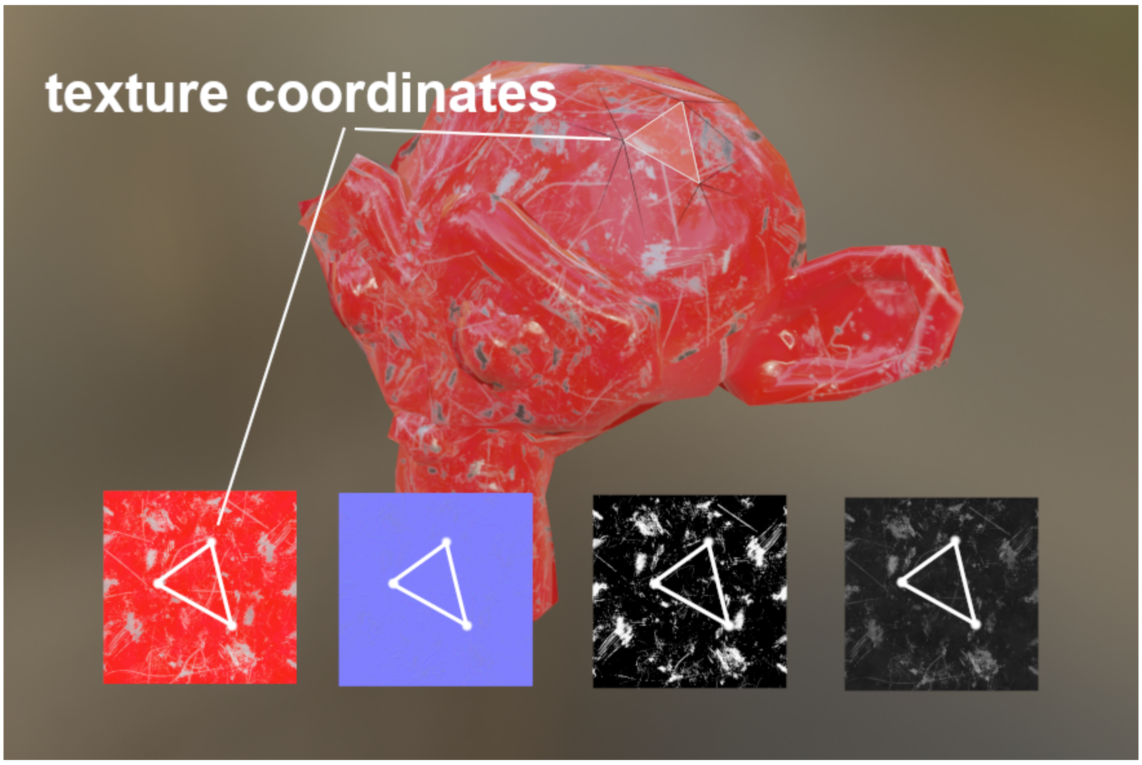

Figure 2.1. *Physically based materials applied on top of Blender Suzanne model. Vertex encoded texture coordinates are used to sample the four PBR textures. From the left, the textures are base color, normal map, metallic, and roughness.*

where, L_{out} is radiance to a view direction, L_e is the amount of *emission*, that is the power emitted of the hit surface, Ω means that the integration is performed over the hemisphere around the point through all the negative incoming angles ω, L_in is the incoming radiance, f_r denotes the *bidirectional reflectance distribution function* (BRDF) and $cos\theta$ the angle between surface normal and incoming light direction.

3D image synthesis is primarily solving this equation for each pixel. Commonly in modern rasterization, the equation's integral is simplified to a sum of the radiance contribution. The contribution is formed from all of the light sources in the scene shading the surface multiplied by underlying color [8, 14]. In Monte Carlo path tracing, hundreds and thousands of paths are shot from the camera through the imagined screen's pixel to the scene, and when the ray touches a surface, it reflects other random directions. Each reflected part of the camera path contributes slightly to the final pixel's color, and when performed thousand times with randomly selecting the directions, the result has more realistic lighting than simple rasterization. The reason for the apparent realism comes from lighting effect called *indirect lighting*, or sometimes called global illumination. It is a subtle effect that comes from light bouncing and reflecting all surfaces. Trying to solve or approximate it has been a research interest for a long time.

Incoming radiance L_{in} for a camera c and a given view ray v can be noted with

$$L_{in}(c, -v) = L_{out}(p, v),$$

where the L_{out} is the outgoing radiance from a surface intersection point p, where the view ray has hit. We assume there are no participating media, like smoke, and we are also interested only in solid surfaces that do not let light pass through it or refract it. Here, radiance can be emitted straight by the surface itself, which is shown in the Eq. (2.1). More commonly, the surfaces do not emit radiance by themselves but reflect some light emitted elsewhere. A given point reflects incoming radiance in the opposite direction. Locally, these surface reflectance phenomena and their subsurface scattering is denoted by BRDF $f_r(l, v)$, where l is the incoming light direction and v the outgoing view direction [15].

When any surface is examined at the micro and atom level, different types of irregularities can be observed. The realistic simulation of light scattering distributions of surface materials have received decades of research attention [16, 17], However, more recently, a *physically based rendering* (PBR) microfacet material model has been re-found and widely adopted, called *Trowbridge-Reitz / GGX* [18, 19]. The GGX model includes a geometry term, a Fresnel term, and a normal distribution term, which combines different physical microfacet phenomena approximations. The geometry term approximates shadowing and masking that happens on a micro surface level. The normal distribution term approximates the distribution of the surfaces normals, and the Fresnel term approximates how reflected light depends on the viewing angle. In addition to viewing directions and light directions, these BRDF terms require only three parameters: how metallic the object is, how rough the surface is and what is the *index of refraction* (IOR) for the object [19]. With these, almost every solid object can be synthesized in the virtual worlds.

Materials are the basic format to pass around the surface specifications in different 3D softwares. They often consist of *textures*, which are images that contain pixel-wise parameters for each of the reflectance terms, and other parameters, like whether the surface should be transparent or emit some light. Textures map pixel-specific roughness and metalness values, but Fresnel is usually a constant for the whole material [19]. The popular metallic roughness has a less physically accurate model than the earlier Phong shading model [20], but it is easier to work with. This workflow allows a convenient mental model for artists to approach authoring GGX materials, selecting whether the material has metallic parts in the metallic texture and how rough or smooth the surface is to the roughness texture. An example of how the GGX workflow textures are mapped on a model is depicted in Figure 2.1. Using *texture coordinates*, we can sample parts of surface triangles and retrieve the used GGX settings with coordinates. Texture coordinates are simply 2D image coordinates.

Virtual 3D scene's geometry can be represented by connecting three *vertices*, 3D points

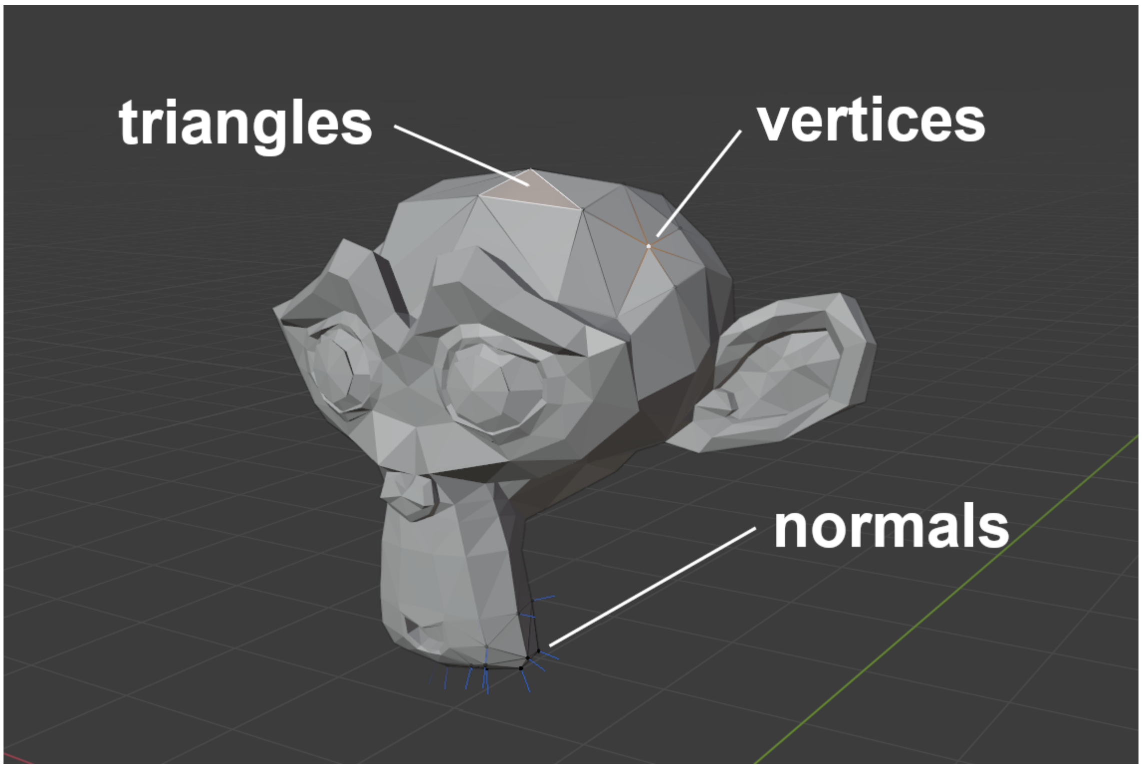

Figure 2.2. *A vertex, a triangle, and a normal perpendicular to surface annotated on Blender Suzanne model.*

in space, together, forming a *triangle* as shown in Figure 2.2. A ray can be imagined shot out of the camera towards the triangle and intersecting it. Color can then be fetched from the intersection point. Moreover, by taking the encoded color of each vertex, a blended color can be formed. Forming is done by weighting with their location differing from the intersection point. If such methods are followed further, any complex scene can be formed by combining thousands and millions of triangles. This is the basis of all 3D graphics [8, 21]. In addition to simple triangles, there are quads, often used in facial expressions and other surface curvatures expressed in mathematical forms. They are commonly described as *faces*, a primitive patch of the surface. In this work, we focus mainly on the triangles.

The next most important primitive after vertex is a normal, a vector pointing outwards of a surface. For a smooth surface, it is perpendicular to the surface tangent on a point. In 3D graphics, they are commonly encoded for each of the vertices or faces in the scene [8]. They allow shading the geometry; by comparing its direction to the camera's view direction and the incoming light's direction, we can determine how much of the incoming radiance should be shaded to the point. The shading normal of a surface is sampled during the BRDF shading. Sampling is used to determine how much light reflects on a surface from light to the camera. If the normal points to the almost opposite direction of all incoming light directions, the point should be not be shaded at all. This would be the case of enclosed surfaces like if a camera is inside a cube and all the emission sources outside, the rendering would be completely black. A commonplace trick in 3D tools is to smooth the normals, slightly altering its direction [8, 21, 22]. This technique lowers the number of faces required to produce smoothly lit surfaces.

There are a couple of standard ways to describe lights in virtual scenes. In rasterization, it is common to have lights described in a singular point, and with path tracing, lights often have some area they sample from [8]. The most straightforward lights are directional lights, which do not have a location in the world, and light direction l stays constant. Then there are punctual lights, which are our main interest in this thesis. Punctual lights have a singular position and no surface area. The two most popular types of punctual lights are point lights and spotlights. Point light's contribution to a surface point depends on the shaded point's location and the location of the lights, as we saw with the Eq. (2.1). Point lights emit light uniformly in all directions. It is common practice to apply a radius with point lights [22]. Point lights with radius can be sampled in the Monte Carlo path tracing. The lights area is sampled, and the probability of sampling that point is used to weigh its contribution to the final radiance amount. This method is called *multiple importance sampling* [23]. In addition to the punctual and area lights, emissive triangles and textures can emit light.

To convert a 3D world to a 2D screen, a common practice is to use different coordinate systems. During the creation of 3D models, one coordinate system, second when the models are moved around the scene, and third when the models are finally rendered. A typical workflow is for a single object to be transformed from local space to world space, then to camera's view space, then to camera's clip space, and finally to screen space [8, 24, 25].

Conceptually, there are five different spaces from which we can transform to and from. In local coordinates, we have vertices of an object relative to its local origin. When a 3D object is created, its vertex positions are relative to the origin of that system. World space means that the observed scene now has multiple objects, and they are now relative to the world origin and have some offset from it [24]. Translating an object to its desired position in the world is done with *model matrix*, which we also call model to transform in this work [8]. Model transform is most actively updated during the run-time of animations and interactive applications, like in games when characters move around the world. If a virtual camera is placed in the world and we want to see what the camera sees, the easiest and common practice is to create a transform matrix that transforms each vertex in the scene so that the camera becomes the center of the world. Such matrix can be formed with cameras extrinsic parameters, like the position and orientation of the camera. This matrix brings the world into the view space. From view space, the 3D perspective can be achieved with a projection matrix that can be created with the camera's intrinsic parameters, which are often in computer graphics described with aspect ratio, near plane, and far plane [8].

The final space conversion is from the view space to the screen. This can be done with a clip space matrix [8, 24]. Mathematically speaking, when we have a clip space matrix M_{clip}, we can transform a world space coordinate p_i by

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = p_i M_{clip}, \tag{2.2}$$

where $x_{clip}, y_{clip}, z_{clip}, w_{clip}$ are the new coordinates in the camera's homogeneous clip space [24]. These coordinates can be brought to a normalised device coordinates with perspective division:

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix},$$

where $x_{ndc}, y_{ndc}, z_{ndc}$ are the coordinates now in normalised device space. Finally, we yield the screen space x, y coordinates by converting the coordinates from range [-1, 1] to [0, 1], and then multiplying them to the resolution of the rendered screen width w and height h:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} (w-1)\dfrac{x_{ndc}+1}{2} \\ (h-1)\dfrac{y_{ndc}-1}{-2} \end{pmatrix}. \tag{2.3}$$

Retrieved screen space coordinates x, y now represent where the vertices end up in a screen [24].

2.2 Animations

Dynamic and interactive virtual world simulation requires transformations. The position of a vertex can be changed with an affine translation with a homogeneous transformation matrix. The standard convention in computer graphics is to multiply translation, rotation, and scale matrices to a single matrix *TRS*, which can be compactly be sent to GPU in vertex shader the final position calculation [8, 25]. In three dimensional, OpenGL like right hand coordinate system, given p as a position vector of a single vertex can be transformed with TRS matrix by:

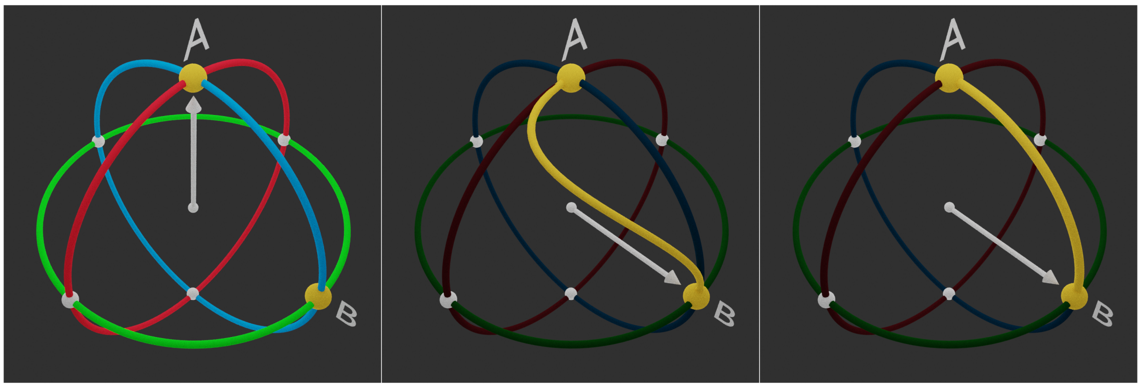

Figure 2.3. *Graphics applications using Euler angle representation apply rotations around three axes one after another. In the middle figure, a yellow curve displays the path from A to B Euler angle representation may take if the rotation is done by rotating the red and green circles 90 degrees. The image on the right shows the shortest path from A to B achieved by rotating with a unit quaternion.*

$$p' = \begin{bmatrix} S_x R_{00} & R_{01} & R_{02} & T_x \\ R_{10} & S_y R_{11} & R_{12} & T_y \\ R_{20} & R_{21} & S_z R_{22} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix},$$

where S is the scaling multiplier for each axis, R is the three-by-three Euler angle rotation matrix, T is the added translation, and p' is the newly transformed position [24]. Transforming all the vertices of an object with a TRS matrix is often called *rigidbody transformation*.

There are two sizable issues with the three-by-three rotation matrices: a gimbal lock and interpolating from one rotation to another. Euler angle representation describes the rotation around three axes. These rotations are pitch rotation, wherein the right-hand coordinate system rotates around the x-axis, yaw rotation, where it rotates around the y-axis, and roll, where it rotates around the z-axis. In Euler angle representation, these three rotation matrices are multiplied one after another. Given that the matrix multiplication is commutative, the order of the multiplication matters, resulting in the lock. The gimbal lock occurs when the last axis in the rotation multiplication chain aligns with the second rotation [26]. One degree of freedom is then lost, as both the second and the third matrices try to rotate around the same axis. The gimbal lock is often solved, for example, in game cameras, by placing the pitch as the last of the three rotations and limiting its angle between (-90, 90) degrees, where it can never align with the second rotation [27]. The range (-90, 90) is acceptable, as it restricts the looking to down on the ground and up to the sky.

The other complication is the interpolation between two rotations. With Euler angle representation, it is non-trivial to rotate along the desired geodesic curve. Even though the start and the endpoints of the animated rotation key frames shown in the example Figure 2.3 are correct, the rotation from point A to point B with pitch, yaw, and roll, does not follow the intended shortest path. Correct curve seen in the rightmost image in Figure 2.3 can be achieved by representing the rotations as *unit quaternions* [28] and applying spherical linear interpolation, shorthand slerp, between two of them [29]. Unit quaternions are a subset of quaternions that represents rotation and can be defined as follows:

$$q = iq_x + jq_y + kq_z + q_w,$$

where q is the unit quaternion, q_w is the real part, and q_x, q_y, q_z the imaginary part where i, j, k are the imaginary units [8]. Imaginary vector part q_x, q_y, q_z of the unit quaternion supports all the usual vector operations, such as scaling, addition, cross product and dot product. Unit quaternions may also be written

$$q = sin\phi u + cos\phi,$$

where u is a three-dimensional vector such that $\|u\|= 1$ and ϕ is the amount of rotation around that vector. With unit quaternion q that when multiplied with its multiplicative inverse q^{-1}, the equation

$$q^{-1}q = qq^{-1} = 1,$$

holds true [8]. The desired shortest arc between the points A and B of the Figure 2.3 can be achieved with slerping from unit quaternion q_a to unit quaternion q_b with software implementation friendly format

$$slerp(q_a, q_b, t) = \frac{sin(\phi(1-t))sin\phi}{q_a} + \frac{sin(\phi t)}{sin\phi}q_b,$$

where $t \in [0, 1]$ is a parameter between the two quaternions [8]. The intrinsic geodesic distance between two unit quaternions that is, the angular change in rotation d_r can be calculated with:

$$d_r(q_i, q_{i-1}) = \|\ln({q_{i-1}}^{-1}q_i)\| \tag{2.4}$$

where q_i and q_{i-1} are the compared rotation unit quaternions [30].

Unit quaternions are handy: it can be shown that a point represented in a vector format $p = (p_x p_y p_z p_w)^T$ is rotated by unit quaternion q

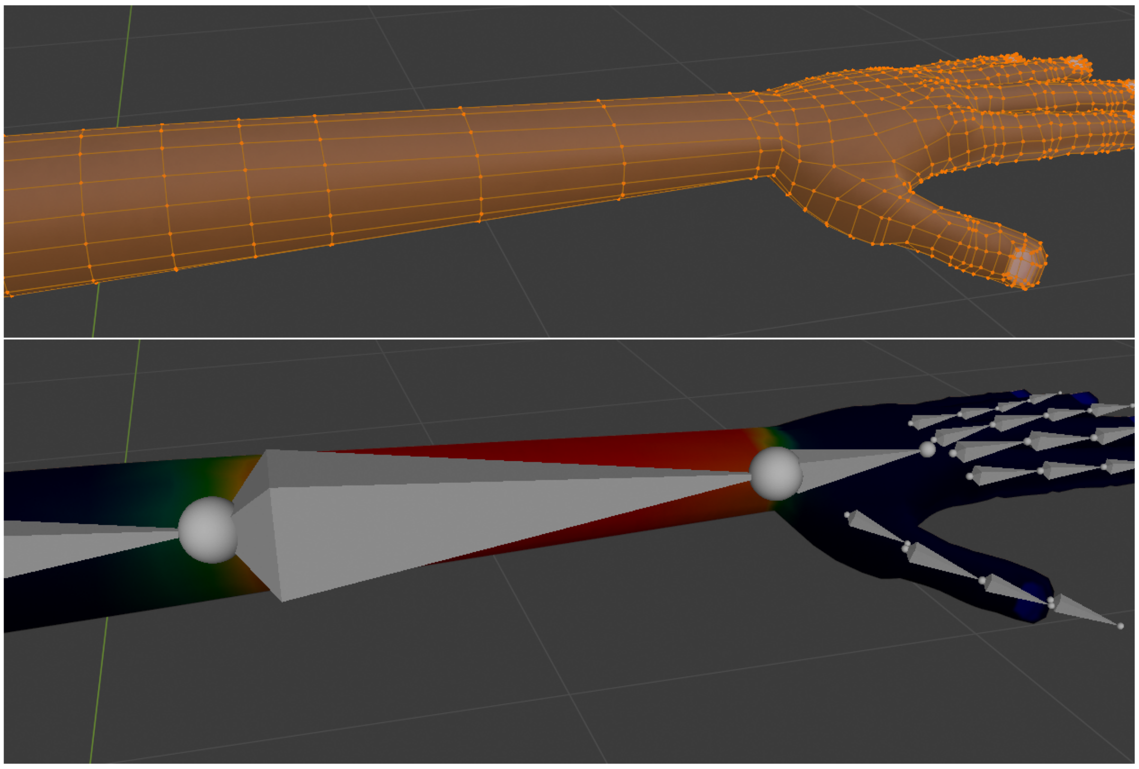

Figure 2.4. *The top image displays skin, and the bottom the skeletal bones and the influence the forearm bone has on vertices. In the bottom image, the red color indicates the joint's total influence, yellow and green that there is some influence, and blue that the forearm joint does not influence the vertex skinning. The character model is Michelle from Mixamo animation library [31].*

$$qpq^{-1},$$

around axis u for angle 2ϕ [8]. When only rotation is required, this four component representation of a rotation is a preferred way compared to the nine component rotation matrix, and it is used in memory-constrained systems, like graphics rendering on GPU [25].

Rotating and translating rigidbodies allows animations to have only rigid motions. In 1988, Magnenat-Thalmann et al. proposed a new technique called *vertex skinning* in the animation world [32], and it has since become standard in all rigging software, 3D games, and animations [33, 34]. This technique has been converted to a practical method. In the method, a skeleton is applied in addition to the mesh to help modify and animate vertices. This procedure simplifies animating characters with human-like motion [35].

Skeletal representation requires two things: a mesh, which is often also called a *skin*, and a hierarchy of bones. A popular name for the combination of the skin and the bone hierarchy is *armature* [22]. When the bones are animated, they control the movement of the corresponding vertices in the skin. In addition to the armature, predetermined animation poses are declared, especially in an interactive context like games. For example, when a game character runs and then jumps, the animations are interpolated between key frames of the running animation and also blended between running and the next jumping animation [33].

A bone hierarchy is represented with a transformation from a *bind pose* [35]. A bind pose, and sometimes called rest pose, is the starting point for the animations, and all transforms are described as offsets from this pose. The skin's vertices must have the information on which joint influence them and with what weight. in Figure 2.4, on the top image, skin is shown, and on the bottom image, a forearm bone and its weights are presented. Vertices under the influence of red-colored areas are transformed fully by the forearm bone, blue-colored areas are not influencing at all, and yellow to green gets some influence from this joint [35]. Bones are in a relational hierarchy to each other. Transforming, for example, a knee of a character will also transform the rest of the foot recursively along with it. The skinning procedure simplifies the animation procedure: only a single rotation matrix is changed, and the whole leg and all of its vertices are moved.

Vertex skinning is more formally called *linear blend skinning*. The vertices are linearly blended near the joints. The world space position for each joint can be computed as the weighted average:

$$p' = \sum_{i=0}^{N} w_i B_i M_i^{-1} p,$$

where p' is the new blended vertex world space position, N is the number of bones affecting the given vertex, w_i the weight the indexed bone influence this vertex, B_i is the local animation transformation for the indexed bone, M_i is a bind pose matrix that transforms bone's coordinate system to world coordinates and p is the original vertex position [8]. Often with human characters, N is four, as it allows most poses and does not require too much data to compute on the GPU [27].

For shading, we also need to update normals. We compute the normals naively with a similar weighted average:

$$n' = \sum_{i=0}^{N} w_i (B_i M_i^{-1})^{-T} n_i,$$

where n' is the new blended shading normal, N the amount of bones influencing the normal, $(B_i M_i^{-1})^{-T}$ the inverse transpose of each blending matrix, in which the M_i is the bind pose matrix and B_i is the animation matrix [35]. However, taking a transpose is inefficient, and the computed normal is considerably inaccurate. A proposed method tackles the issues and achieves better quality normals and more precise results [36]. An example implementation of simple skinning in the vertex shader is presented in the

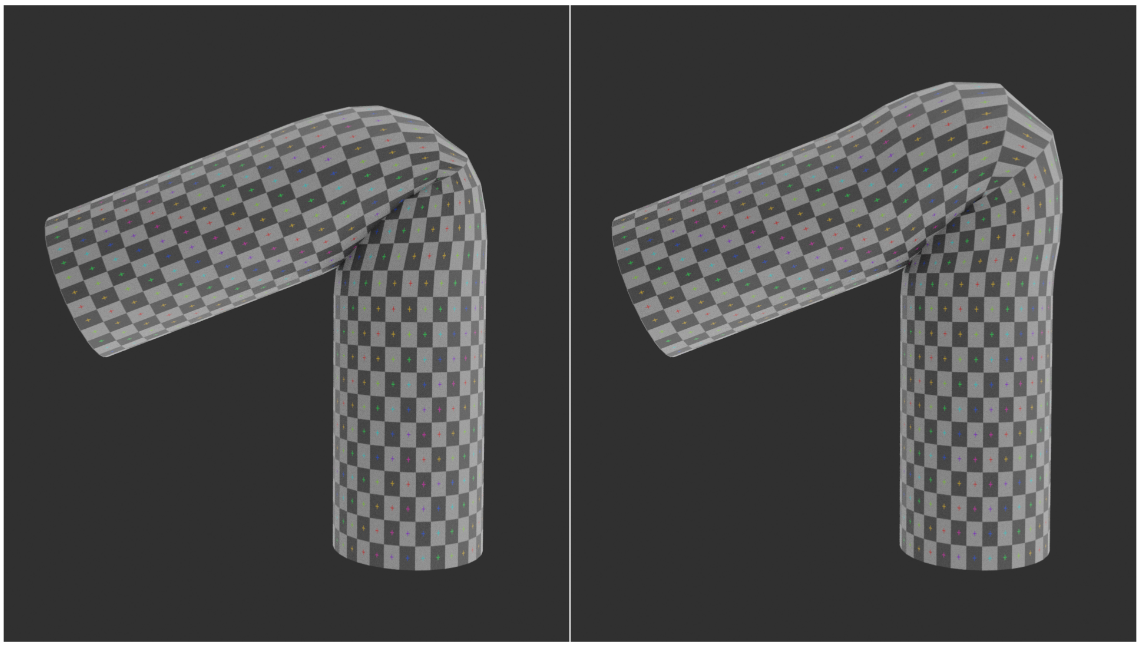

Figure 2.5. *Linear blending (left) and dual quaternion blending (right). Dual quaternions have a lot fewer skinning artifacts compared to linear blending. In linear blending, with extreme angles, the mesh collapses, whereas in the dual quaternion blending, the volume is preserved, resulting in a bulging effect.*

appendix A.1 that is applicable in modern graphics API Vulkan [10].

Linear blending has an issue with extreme blending angles where the geometry collapses. The issue can be observed on the left cylinder in Figure 2.5. The issue can be fixed with *dual quaternions* [37]. They extend the basic unit quaternion with a concept of dual numbers. Dual numbers are expressions in the form of

$$d = a + b\epsilon,$$

where d is the dual number, a and b are real numbers and ϵ is the complex symbol to satisfy $\epsilon^2 = 0$ [37]. Quaternions are already basically dual numbers, with their real and imaginary parts. Translation can also be converted to dual number:

$$d_{translation} = (1, 0, 0, 0) + \frac{(0, x, y, z)}{2}\epsilon,$$

where $d_{translation}$ is the translation in dual number, and x, y, z are the coordinate change in the vertex position. This dual number representation of a translation can be combined with a rotation to a single, dual quaternion:

$$d_{transform} = d_{translation} d_{rotation} = q_{rotation} + \frac{(0, x, y, z)}{2} q_{rotation}\epsilon,$$

where $d_{transform}$ is the dual quaternion that can perform both the rotation and the translation, $d_{translation}$ is the translation in dual number form and $d_{rotation}$ is the rotation in dual number form [37]. Dual quaternions can be decomposed back to the rotation in quaternion format and the translation in vector format [37]. When blending is performed with dual quaternions, the volume of the skin is preserved, as seen on the right cylinder in Figure 2.5. Dual quaternions are also quicker to blend, with generally over 20% improved performance, and they take only eight components, compared to 16 that TRS matrices take, making them perfect for skeletal animations [37].

In addition to rigidbody and skeletal animations, the final animation method is to animate each vertex explicitly. This method is called *vertex morphing*, and sometimes the name shape keys animation is used [8]. Morphing is most commonly used in facial animation, where each key frame of the animation has new positions to all vertices, which are interpolated between [38].

3 TEMPORAL RENDERING

Rendering workload can be lowered by taking advantage of the fact that subsequent frames are very similar. In this Chapter, we look at temporal reuse algorithms utilizing the frame-to-frame image coherence to speed up the rendering. Then we define requirements for an excellent benchmarking dataset for this domain. In the following section, we present metrics for comparing how temporally challenging different datasets are for rendering. Finally, we review the previous methods used for capturing animations.

3.1 Reuse Algorithms

There is a body of research done both path tracing and by the real-time constrained rasterization and ray tracing to reuse temporal data to have better performance [39]. Given the time complexity of rendering an image, shortcuts are often taken to lower the count of sampled paths and the length of the path in the Monte Carlo integration or by lowering the sample count of spatial anti-aliasing methods like Multisample Anti-aliasing (MSAA). The continuous image function must be sampled in high frequency so that the final pixel receives anti-aliased color. Recent survey identified the two components of *Temporal Anti-aliasing* (TAA): sample accumulation and history validation [40].

The figure 3.1 shows a generalized execution flow of the TAA algorithms. Each frame, the renderer first streams necessary image features to few clever steps. *Motion vectors* are the pixel's velocity during the animation, which can be used to reproject where the pixel was in the previous frame. A color sample, a returned color value of a single camera ray, is validated against the reprojected history by utilizing other features. With the knowledge of whether the history is valid or not, it can be rectified. Finally, the new sample is accumulated on top of the history. Then, it can be sent for the next frame to use and post-processing and display.

In TAA, different sample positions are selected for each frame that aligns temporally to a supersampled result, which amortizes the cost of supersampling [41]. Sample positions are jittered with a sampling strategy, state of the art being low discrepancy patterns like Halton or Sobol sequence, which evenly distributes samples and mitigates problems if the temporal integral should be restarted [42, 43]. Amortized pixel's history buffer is shown in Figure 3.2, where new sample positions are accumulated in each frame. The exact

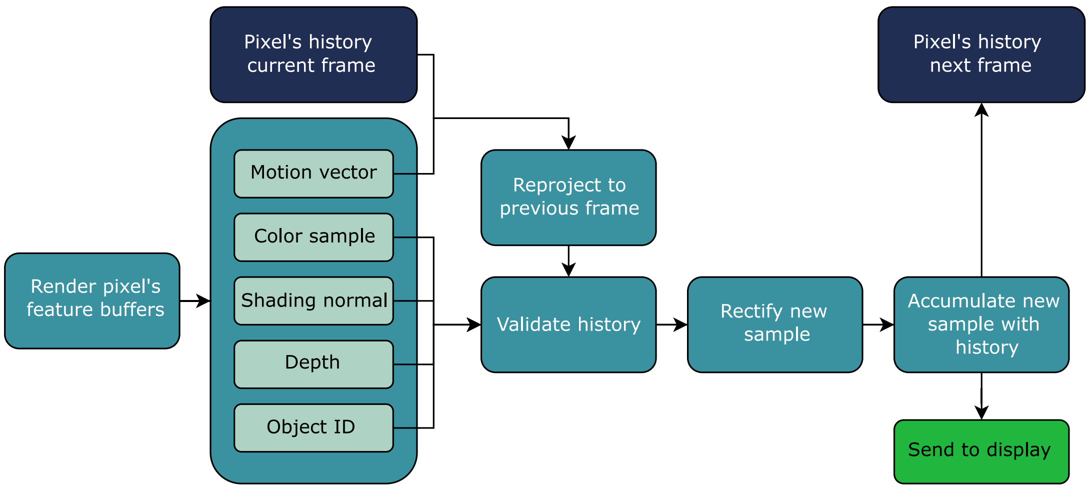
Pixel's history current frame
Pixel's history next frame
Motion vector
Color sample
Shading normal
Depth
Object ID
Render pixel's feature buffers
Reproject to previous frame
Validate history
Rectify new sample
Accumulate new sample with history
Send to display

Figure 3.1. *The diagram shows a typical flow of execution and used procedures in temporal reuse algorithms, like TAA. First, the feature buffers are rendered. Then, using the history buffer of the previous frame, the history of the current pixel can be validated, and with a new sample, the history might be rectified. After that, the new sample is blended with the history colors. The final color can be sent to the screen and for the next frame to use.*

process works on other methods requiring sample integration. These effects include diffuse global illumination, ambient occlusion, shadows and reflection [40].

After a pixel has been reprojected and history retrieved, new samples are then constantly accumulated and blended together:

$$f_n(p) = \alpha \cdot s_n(p) + (1 - \alpha) \cdot f_{n-1}(\pi(p)),$$

where $f_n(p)$ is current frame n's color at given pixel p, α is the blending factor, $s_n(p)$ is frame n's new sampled color at pixel p and $f_{n-1}(\pi(p))$ accumulated pixel history from previous frame, that has been reprojected with reprojection operator $\pi(\cdot)$. The blending factor can vary the rate the oldest samples are forgotten from the color history.

Reprojecting a pixel is simple if not the scene nor the camera have changed from the previous frame, but with the dynamically changing scene, it is not as easy. Common practice is to use *reverse projection* displayed in Figure 3.2 [41]. Similar but the opposite approach is to forward reproject, where the previous frame's pixels are projected on the current frame, but it is less efficient on graphics hardware [44]. The known motion of the pixels movement, motion vectors, are used to sample where the current pixel was located in the previous frame and accumulated accordingly. It is common practice to use hardware-accelerated bilinear texture fetch when retrieving the previous color from the history color texture in a real-time context. The fetch mitigates issues compared to naive reprojection, as it might result in in-between pixels and camera sub-pixel offset that,

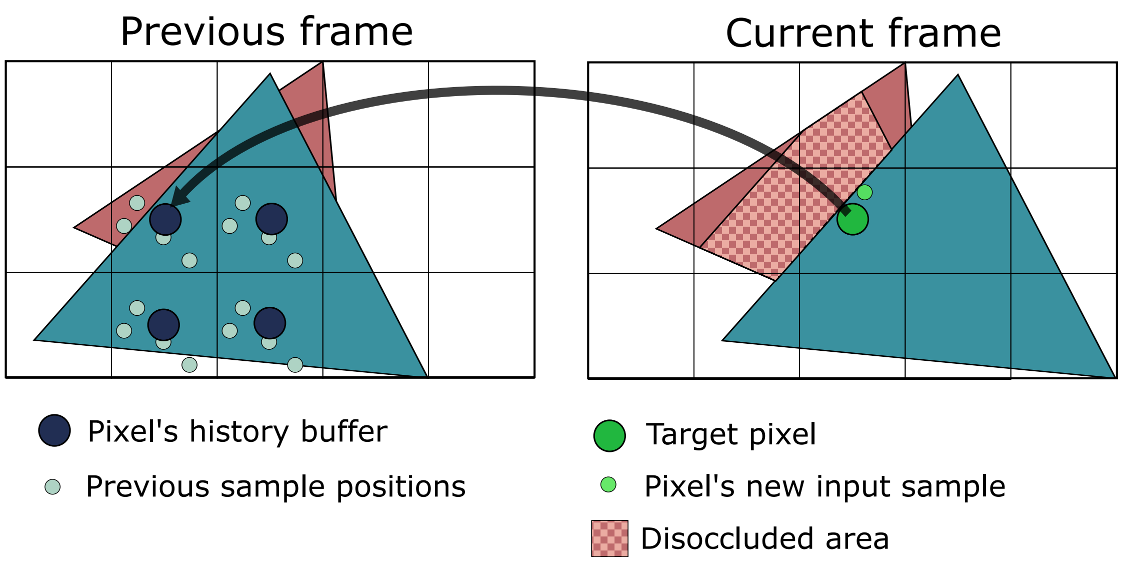

Figure 3.2. *Sampling the history of a target pixel. The history buffer (left) has been formed from previous sampling positions. Often, these sampling positions are not saved separately but accumulated as the final color of the previous frame. On the current frame (right), a new input sample is taken from the image blended with a bilinearly sampled history of the previous frame's pixel position. The disoccluded red area appearing behind the blue triangle would require more samples to affect the final pixel value.*

due to the *jittering*, results in non 1:1 mapping between the previous and the current frame. Jittering means that the sample position is offset from the center of the pixel. In the bilinear texture fetch, the wanted color is interpolated from four nearby samples and weighted accordingly as seen in Figure 3.2. In motion, this has a downside of introducing *resampling blur*, as it softens the resampled image. Resampling happens in each frame so that the high-frequency detail can be quickly lost.

High-quality motion vectors are not trivial to come by with [45]. Often game engines create a motion vector texture that has the offset between current and previous pixel's location by transforming the geometry twice with current and previous camera matrices [46]. Problems arise when shading is not geometry related, which is the case with transparent and highly reflective materials, lights, and shadows [45].

Another issue with sample accumulation is the reintroduction of aliasing artifacts for already smooth edges on moving objects. There are few methods proposed to solve this, for example, using a 4-tap dilation window or some other adaptive filtering schemes [47]. With too small blending factor α the color may start to have resampling error or temporal lag. To avoid such issues, history is ensured to be refreshed by clamping the α to lower value. Motion can also be used to adapt the α so that the bilinear resampling error is prevented. In rasterization, there is also an issue with sampling and integrating over pixel area in both geometry and texture, and because textures are usually filtered with mipmap methods, the result might be overtly blurred. Typically a mipmap bias is applied, which also helps to solve issues introduced with *temporal upsampling methods*, that we will describe shortly [46].

As the camera and the world move, the lighting conditions change, or there might appear disocclusion like in Figure 3.2. The validity of the history data should be rejected if an error is detected. This history's confidence can be mainly detected from two sources, either from geometry data, like depth, normal, motion vector, or object ID, or from the color or radiance data. Disocclusion can be recognized with geometry data by comparing, for example, the current pixel's depth value with the pixel history's depth value, and if they are further away from each other than a small scene dependant error tolerance, the pixel can be marked incorrect [44]. Even more robust matching can be achieved with additional geometry data, like surface normal or object identifier. The whole history may not need to be discarded, though, as we will discuss history rectification strategies later.

With shading changes, like shadows, lighting, and reflections, geometry data is not sufficient. Checking the validity of the shading can be done efficiently with color and radiance data [48]. Directly comparing color indicates whether the data is invalid because of visible light or shading change and indirect motion vectors. However, comparing aliased the current frame's samples to anti-aliased history might result in bias of our error estimation, so there have been few methods proposed to help in the comparison [41].

Rejecting invalid history resets the temporal integration process, which may lead to artifacts [41]. Rejected data can be made more consistent by taking more samples. This is called *history rectification* [49, 50, 51]. The idea is to pull the color information from the neighborhood to our sparse and aliased samples. With the neighborhood colors, a color bounding box is created, and if the current sample falls inside it, history can be connected and rectified. Otherwise, it should be discarded. When the rectified history is blended with these artifacts are removed. More sophisticated color neighborhoods and gradients have been proposed to have better quality rectification [14].

Rectifying the history of a pixel also comes with downsides. Temporal artifacts, like ghosting, might appear when the history is not invalidated correctly and forgot [14]. Rectification techniques assume that the neighbourhood of the pixels surface point contain similar values. Since there are only a few samples for the pixel, highly detailed content with a thin geometry or shading feature is easily missed, which results in dropping the pixel's history. Without temporal supersampling history, the quality suffers from aliasing or temporal instability issues, like flickering.

Temporal upsampling is another technique to achieve a higher frame rate or resolution [41, 52, 53]. The technique's benefit is that the sampling rate is reduced from one sample per pixel to some portion of a sample per pixel. Upsampling results in higher resolution images by just accumulating lower resolution shading results. In practice, the input samples are upscaled to desired output resolution and then normalized to it. Normalization can be done with the sum of the weights with a neighborhood averaging filter, like a Gaussian kernel.

Like with all of the reviewed temporal algorithms, there are few issues with the upsampling methods. When upscaling is applied to the sample accumulation function, the high-frequency detail is lost [14]. With the temporal jittering strategy, each upscaled pixel receives now and then high-quality samples that have not been interpolated from the nearby upscaling neighborhood, resulting in temporally suitable quality pixel. Also, a confidence quality factor for each output pixel has been proposed [46]. With it, the history is retained when the quality of the accumulated input sample has low confidence. The upscaled sample accumulation also suffers from the same problems that we mentioned before, and now there are even fewer input samples to help solve the issues. History cannot be easily rectified, which leads to carefully selecting between ghosting, temporal instability issues, like flickering, and blurriness.

The three-by-three neighborhood used in rectification for the color bounding box computation is now larger than the three-by-three multiplied by the upscaling multiplier, for example, nine-by-nine. A few different approaches have been used to help in the rectification process. In a proposed method, they compute a smaller neighborhood that is based on the used sub-pixel offset for the color bounding box calculation [46]. The temporal ghosting artifact is reduced, as the sampled points further away from the pixel are not blended. Similarly, a fixed two by two neighborhood has been proposed, with complimentary results [52]. One other method used in the game Quantum Break trades the quality of dynamic lights, shadows, and animated textures to have sharper and more stable static shots [54]. Their method uses motion vectors' speed to relax the color bounding box and uses a tighter bounding box with smaller vectors to increase the sample accumulation.

An interesting recent finding with temporal upsampling is the noticeable differences when the order of the sampling interacts with dynamic geometry [52]. They recognize two modes for the sampling, bow tie, and hourglass, which work better when sampling motion is horizontal or vertical. In addition to these, they show how increasing the rendering frequency lowers the required temporal sample count, as it uses the human visual aid system to integrate the samples into a perceptually sharper image with fewer visible artifacts.

A similar thing to upsampling is *checkerboxing* (CBR). In CBR, instead of applying jitter offset on the sampling locations, a diagonal checkerboard pattern is used. The method integrates temporally higher resolution images. These CBR methods often utilize a hardware dependant and accelerated sampling strategies. It was widely used in previous generation game consoles to achieve a unified upsampling strategy to reach 4k resolution [55]. Given that only portion of the pixels are shaded, the history must be used to fill in the gaps. The CBR method suffers from the same problems that the upsampling method does.

In the virtual reality rendering *motion sickness* [56, 57] is still a common problem, but temporal methods have been utilized in aid for it [4, 58]. In motion sickness, the user faces physical discomfort caused by perceiving visually conflicting information to what they experience. Practical solutions to this include lowering the latency, increasing the frame rate, and using high-quality motion tracking [59, 60].

The temporal data can approximate the result when the rendering does not finish in time [58]. Reprojecting the previous image to a new position when a user turns their head is called *asynchronous reprojection*, or time warp. Practically, each frame is reprojected asynchronously along with the actual rendering, and when the rendering time exceeds, the prepared reprojection can be used instead [61]. VR rendering also has temporal complications that are present when the scene changes a lot. *Color fringing* is an issue where colors smear to each other in some VR headsets as each of the colored lights, red, green, and blue, are temporally displayed one after another [4]. Then there is *judder*, that appears with quick head movements. In judder, high persistence and low refresh rate introduce a motion blur-like effect, where pixel colors are both smearing and strobing [62]. The quick head movements are the problem with the VR: even with a rapid refresh frequency of 60 frames per second, a human can turn their head several hundred degrees per second, and the same time gaze in the opposite direction, doubling the effect [4]. Furthermore, there is a complicated problem that comes from natural human motion: when the head is turned, but the gaze is fixed on a point, humans are used to seeing crystally clear the point they are focusing on with their eyes. However, this may not be the case, as the temporal methods may introduce some motion blur.

Deep learning in rasterization has also recently become of interest in temporal reuse cases. *Deep Learning Supersampling* (DLSS) 2.0 by NVidia builds up from the TAA ideas of upsampling and temporal reuse by applying deep learning with it [63]. The network is taught with high quality ground truth images to learn the fundamentals of reconstruction. The algorithm has not been released to the public yet, but in the presentation slides, Liu et al. claim DLSS 2.0 to be using as input features the sampling jitter offsets, geometric motion vectors, depth buffers, low-resolution pixel samples, and exposure. Many of these are familiar feature buffers to TAA. There are no research comparisons yet with the quality of this rendering method compared to the previous state of the art, but the public reception about the perceived improvement in image quality and rendering performance has been overall positive [64].

There are not too many temporal reuse methods in the world of path tracing, as there is infinite time to produce the image. The recent development of trying to render the image with path tracing in real-time has introduced some solutions and problems. These include using small sample counts and spatio-temporal features to denoise the image [65, 66].

Even though these methods converge to clear images with global illumination with static scenes, when temporal motion is introduced with the camera, with the scenery motion, or the lighting conditions change, they start to blur, and the global illumination begins to lag behind [66].

3.2 Benchmarking Temporal Rendering

Benchmarking helps to better understand the underlying problem and identify parts that still require improvements. In this chapter, we first discuss some of the requirements for a satisfactory dynamic benchmark. Then, we describe comparison metrics that can be used to compare temporal features of dynamic datasets, and finally, we review methods that have been previously used in the capturing of 3D animations.

3.2.1 Benchmark Requirements

We are planning to publish the benchmark to be openly used in future graphics research. The publication plan gives us some explicit constraints for the file format selection and the content to be included in the dataset. We recognize that the dataset should have

- focuses on the temporal content,
- dataset is delivered in a simple and easy to use format,
- dataset is released with a permissive licence,
- it contains quickly moving camera, 3D scenery and lights,
- its material model and the geometry complexity are modern,
- it uses skeletal animations.

Having a clear focus and using a single file to provide the dataset attracts users by having low overhead to try it. We aim to improve upon the previous work in the temporal qualities, but it should also have similar complexity in geometry and materials to previously released sets, like modern ORCA datasets [67]. ORCA datasets have relatively high resolution in their PBR textures. Therefore, the dataset format should also support and use PBR textures it. We also plan to capture the animation from a game, so the file format should also support skeletal animations. Next, we will review some of the previously used 3D capturing methods.

3.2.2 Dataset Comparison Metrics

We recognize the following errors and downsides in the currently used temporal reuse algorithms. These issues are blurriness, ghosting and temporal lag, temporal instability, and under-sampling artifacts. We aim to create a benchmark dataset that introduces situations where these problems are brought forward and where they can be easily examined. In a typical interactive scenario, like in games, the *temporal coherence* can be noticed in the majority of the screen [68]. Temporal coherence means that the pixels are precisely the same as in the previous frame. To compare the challenge our proposed dataset has temporally compared to previously released benchmarks, we measure the opposite of the temporal coherence: the percentage that should be discarded as invalid. We focus on the same features many of the temporal methods use as their inputs and compare how they change frame-to-frame. Namely, these features are the distance a pixel changes its position in the world, the shading normal used in lighting calculations, and the direct and indirect radiance the pixel emits towards the camera. Next, we will go through how these metrics are formed.

The first metric we look at is the movement of the camera through the scene. The camera moves in XYZ coordinates and rotates around its center point. We calculate the distance d_p the camera travels each frame:

$$d_p(p_i, p_{i-1}) = \parallel \overrightarrow{p_{i-1}p_i} \parallel, \tag{3.1}$$

where p_i is the position on this frame, and p_{i-1} the cameras position last frame. To compare camera orientation, we can calculate the difference between the current frame's and previous frame's unit quaternions with the Eq. (2.4). In addition, we determine per axis rotation using

$$d_r(\delta_i, \delta_{i-1}) = \parallel \delta_i - \delta_{i-1} \parallel, \tag{3.2}$$

where δ_i is the angle in pitch, yaw, or roll rotation for the current frame, and δ_{i-1} the angle for the previous frame.

To receive the previous frame's history buffers, we have to reproject the current pixel to the previous frame. We can do this simply by projecting the current frame's world space position with the previous frame's camera's clip matrix, and convert to the previous frame's screen coordinates using the camera transforms shown earlier in the Eq. (2.3). Given the screen space coordinates x, y, we can discard those pixels that reside outside of the image's edges. We determine whether the pixels are outside of the frustum with a discard function $f_{frustum}(x, y, w, h)$:

$$f_{frustum}(x, y, w, h) = \begin{cases} 1, & \text{if } (x < 0) \parallel (w - 1 < x), \\ 1, & \text{if } (y < 0) \parallel (h - 1 < y), \\ 0, & \text{else}, \end{cases} \tag{3.3}$$

where x, y are the reprojected screen space coordinates and h, w are the height and width of the screen. We apply it to get the discarded percentage $f_{percentage}(w, h)$ per image by:

$$f_{percentage}(w,h) = \frac{\sum_{i=0}^{w}\sum_{j=0}^{h} f_{discard}(x_i, y_j, w, h)}{wh}, \tag{3.4}$$

where x_i, y_j are the reprojected coordinates retrieved with indices i, j running through the size of the image's w width and h height. Finally, we calculate the mean of the discarded pixels through the length of the animation with:

$$\frac{1}{N}\sum_{i=1}^{N} f_{percentage_i}(w,h), \tag{3.5}$$

where n is the number of frames in the animation.

Next, we introduce a discard function for each of the feature buffers, where the current frame's values are compared to reprojected values. For each reprojected feature buffers we also use bilinear sampling for the retrieved reprojected pixels [69].

First, we discard by the appearing disocclusions and occlusions using the reprojected pixel's world positions. These pixels have invalid history, as they appear behind moving object, as was seen in Figure 3.2, or moving object occludes previously static pixels. We form a metric, similar to the depth-based edge-detection estimator by [48]. Using current frame's world position vector p_i and previous p_{i-1}, we compare the distance of the two and discard those that are too far apart:

$$f_{worldPosition}(p_i, p_{i-1}, a_{worldPosition}) = \begin{cases} 1, & \text{if } \| \overrightarrow{p_{i-1}p_i} \| > a_{worldPosition}, \\ 0, & \text{else,} \end{cases}$$

where $a_{worldPosition}$ is a discard threshold value. This value is similar to the threshold used by [48] in their transfer function, where we want to create a discard mask over all the pixels whose distance is too far apart from each other. We have tuned this threshold for each scene to mitigate the effect that each of the scenes is a different size. For example, a small room should have tighter thresholds compared to a small city. Scenes can also have been arbitrarily scaled to non-reasonable units.

Threshold selection affects a lot of the number of pixels being discarded. With world position and normal feature buffers, we use similar values for each of the scenes, and with direct and indirect lighting, we tuned the value so that the effect of the noise in Monte Carlo path tracing would be minimized. Tuning the threshold due to direct and indirect light is similar to the exposure control in cameras, where scenery with direct sunlight and indoor lighting requires different settings when capturing photos. Each of the discard functions can also be used with the Eq. (3.4) to get the screens to discard percentage and taking the mean of that with the Eq. (3.5) to have an idea of the average discard throughout the whole animation.

We know from the rendering equation Eq. (2.1) that the final color is linearly affected by the pixels' shading normal. Too big of a change in the normal may result in a different color, and we should discard it as a low confidence pixel. The discard procedure with normals is similar to that with disocclusions: we use the current and the previous frames' shading normals n_i and n_{i-1} and their vector distance to construct a discard function $f_{shadingNormal}$:

$$f_{shadingNormal}(n_i, n_{i-1}, a_{shadingNormal}) = \begin{cases} 1, & \text{if } \| \overrightarrow{n_{i-1}n_i} \| > a_{shadingNormal}, \\ 0, & \text{else}, \end{cases}$$

where $a_{shadingNormal}$ is again a scene dependent threshold value.

Observing the incoming light of a pixel in the rendering equation (2.1) for the first light bounce can be thought of as direct light. The emissive surfaces L_e of the first bounce often contribute most to the lighting, and if a powerful light, for example, the sun, is visible by the pixel, its power has the most significant effect on the final color. We are interested, for example, if the lights are no longer visible by the pixel, the area should be shadowed. To compare these direct light changes, we form a direct light discard function f_{dir}:

$$f_{dir}(L_{dir_i}, L_{dir_{i-1}}, a_{dir}) = \begin{cases} 1, & \text{if } \| \overrightarrow{L_{dir_{i-1}}L_{dir_i}} \| > a_{dir}, \\ 0, & \text{else}, \end{cases}$$

where L_{dir_i} is the pixel's direct radiance this frame and $L_{dir_{i-1}}$ is from the previous frame, and the a_{dir} is a scene dependant threshold value.

Finally, we observe the indirect radiance the pixel receives, which results from light reflected multiple times in the scenery before hitting a pixel. When the lights or the scenery change, the light may bounce differently around the scene. To compare this indirect radiance change, we create a discard function f_{indir} for the current frame's pixel:

$$f_{indir}(L_{indir_i}, L_{indir_{i-1}}, a_{indir}) = \begin{cases} 1, & \text{if } \| \overrightarrow{L_{indir_{i-1}}L_{indir_i}} \| > a_{indir}, \\ 0, & \text{else}, \end{cases}$$

where L_{indir_i} and $L_{indir_{i-1}}$ are the indirect values current and previous frame and a_{indir} is a discard threshold value tuned for that scene.

3.2.3 Animation Capturing Methods

Interactive virtual applications, like games, have many animations that provide temporal movement. Capturing this movement would provide an accessible way to create benchmarking datasets. However, to the best of our knowledge, there have not been any capturing methods published that captures the animation as it happens in a 3D world and records it to a general animation file. Still, similar methods, tools, and techniques exist that could be utilized in the animation capture.

Standard capturing tools are the frame debuggers, which idea is to add tracing and recording layer in between the application and the graphics API [70]. GPU manufacturers and API providers each have their tools for this: NVidia NSight, Windows PIX DirectX12, Intel GPA framework and Apple Frame Capture Debugging Tools [71, 72, 73, 74]. In addition to these, there are open source tools Renderdoc [70] and Apitrace [75]. These tools record every call and their parameter values from the rendering application to the API in practice. With such information, the debugger can examine how the frame was created step by step, displaying every event that happened.

In order to capture longer animation, however, these tools are not very practical. They support only capturing a small number of frames, and each capture halts the program for a short moment. Moreover, applications often apply *occlusion culling* and *level of detail* techniques before sending geometry to the GPU. Occlusion culling means getting rid of rendering that is not seen by the current camera [76]. In the level of detail, the closer the object is to the camera, the more defined geometry is required [77]. So the resulting capture would have varying levels of geometry details, whereas the highest level would be desired. These methods are examples of how only optimized geometry is rendered, making it impossible to render the whole captured animation.

Capturing only the model transforms to animation has been supported by game engines like Unity and Unreal Engine [78, 79]. In Unity, the feature is called *GameObjectRecorder*. It allows users to record frame-by-frame properties of game objects during the runtime [78]. In Unreal Engine 4 this feature is called *Take recorder* [79]. These can be used, for example, in motion capture or pre-recording physics simulations, to be later used in cinematics [79].

Some deep learning research is done on similar data to 3D rendering, mostly in motion flow research datasets. Motion flow datasets "Play for benchmarks" and "Play for data" modified games GPU shaders and captured required buffers [5, 6]. In the same manner, a similar approach was made with MineCraft, where camera and feature buffers were captured [80].

4 RELATED WORK

In this Chapter, we review previous work done for rendering benchmark datasets. First, we look at processors' rendering performance benchmarks and then publicly available datasets with animations that could be used in benchmarking temporal methods' quality. Finally, we introduce Toasters and ORCA datasets that we use to compare our proposed benchmarks.

4.1 Rendering Performance Benchmarks

Benchmarking the performance of microprocessors has standarized it's place in ever ongoing continuum of achievements in semiconductor industry [81]. CPU and GPU benchmarks test how quickly different processors solve predetermined tasks. We survey next both commercial and open source benchmarks that has been created to benchmark rendering capabilities.

Commercial benchmarks are designed to test different workloads to help consumers and industries in their processor purchase choices. There are rendering benchmarks by SPEC, UL benchmarks, Unigine game engine benchmark and Cinebench by Maxon [82, 83, 84, 85]. Rendering benchmarks takes in scene descriptions with camera settings, geometry, material, and animation descriptions and outputs a performance score. Some benchmarks also output how long rendering each step took and images or a video of the rendering. For example, in UL benchmarks, they have an extensive test suite, with stress tests and feature tests for mobile, PC, and VR. The VR tests, for example, tests whether the computer is capable of supporting VR. They consist of different scenes and settings and make standard game settings available for testing out. They measure frames per second performance, processor bottlenecks, and use a scoring system. In some tests, they test out command queues and run on Compute and Graphics pipelines. Different sets have different graphics enabled like ray traced reflections, tessellation, transparent materials, or particle simulations.

There are a few open-source benchmarks processor benchmarks released that test rendering performance. PARSEC has benchmarks for graphics-related workload, like the FACESIM which tests rendering with skeletal animation-like bone weights and the RAYTRACE testing ray tracing workload [86]. Splash-3 is designed to similar benchmark workloads with PARSEC, and GraalBench was released to test the performance of mobile processors, both also having graphics workloads [87, 88]. These differ from commercial benchmarks in that they have the source code and necessary assets available.

4.2 Rendering Benchmarks

The camera and the scenery should change so that the renderer would have reuse opportunities and challenges. In research, instead of using standard graphics benchmarks, they create a small, simple animation for the given problem and do not release it to the public. We review the previous work with temporal elements that could fulfill this research rendering use case.

A Benchmark for Animated Ray Tracing (BART) was released in 2001 [89]. The benchmark has three scenes, Kitchen, Robots and Museum, that are described with Animated File Format, which extends Neutral File Format by adding animation properties [90]. The test suite has been released with benchmarking purposes to measure ray tracing performance and has been used in dynamic ray tracing research [91]. Each scene is designed with a specific stress goal in mind. The Kitchen scene has considerable differences in the details' density, memory cache performance with the inclusion of hierarchical and rigid-body animations, and varying frame-to-frame coherency in the animations. The Robots scene focuses on the hierarchical animation, distribution of objects in the scene, and bounding volume overlapping. The final stress test is the Museum scene that focuses on the efficiency of ray tracing acceleration structure rebuilding. They also propose methods to measure and compare error when datasets are used with ray tracing algorithms. They propose a frame quality comparison with APSNR, the average of PSNR through the animation, and a rendering performance comparison scheme, where the computer's computational efficiency is minimized.

Moana Island Scene is a complete animation description dataset to an island featured in the 2016 Disney film Moana [92]. Disney provides a production-ready package to render the shot in their proprietary renderer Hypherion, and an additional PBRT research renderer version of the scene [21, 93]. It highlights some of the typical challenges in current path tracing animation production. In the dataset, they mention a challenging amount of geometry and complex volumetric light transport. The dataset is massive compared to other released datasets: the unpacked animation file size is over 131 GB, and there are over 15 billion primitives in Moana's Motunui island. The release of the scene from a highly treasured IP's had its complications [1]. Nevertheless, it has seen some appreciation, as there has already been researched on how to manage and render the scene with interactive ray tracing [94]. There are animations described only for the procedural ocean and few slowly moving camera runs.

MPI Sintel Flow Dataset is a motion flow research dataset created from open source 3D animation short film, Sintel [7, 95]. Sintel is a short fantasy animation. Blender Foundation produced the animation as an open movie. The benchmark dataset contains similar feature buffers to those used by temporal rendering algorithms. These rendered buffers and described camera intrinsic and extrinsic parameters are shared open source, but actual animation and geometry data files are behind Blender Cloud paywall [96]. These datasets do not come with 3D animation files, just the rendered frame images, so they are not compatible with our dataset.

In addition to the Sintel, the Blender Foundation provide many openly available datasets in their demo files [97]. The purpose of the demo files is to display different Blender rendering features. Few datasets could be utilized in animation rendering benchmarks, like the few animations designed for Blender's rasterizer Eevee Wanderer, Temple, and Ember Forest. There are also demos for physics simulations with animation, like the Lava animation. All of the demo files are distributed in blend file format and have varying licenses. However, none of these scenes have vastly moving geometry or cameras, so these sets were not taken to the comparison.

KAIST Model Benchmarks have animated fracturing objects, cloth simulations, and walking animated characters [98]. Similarly, the UNC Dynamic Scene Benchmarks have animations of breaking objects and non-rigid object deformations [99]. The downside of these datasets is the lack of temporally challenging scenarios. They either have slowly moving cameras, aged material models or do not contain moving lights and objects.

4.2.1 The Utah 3D Animation Repository

The Utah repository collection was created in 2001 by Ingo Wald [100]. Wald has previously done a body of research on ray tracing dynamic scenes. He has worked on Intel's Embree ray tracer and currently working at NVidia on RTX, which is a GPU with ray tracing and acceleration hardware [101]. The datasets were released along with two research articles focusing on dynamic ray tracing [91, 102]. His motivation behind setting up a repository for the animations was to encourage more research on ray tracing for interactive applications. He mentions the famous Stanford 3D scanning repository as an inspiration [103]. We selected the Toasters dataset shown in Figure 4.1 to compare with our animation.

Toasters dataset represents well how most of the released benchmark datasets are like, like the UNC Dynamic Scene Benchmarks [99] and KAIST Model Benchmarks [98], with no camera setup and vertices morphing around the scene. The datasets consist of key frames as OBJ files, and their materials described with MTLs. We added a camera that is pointing to the scene as is in the images in the repository. For the comparison renders, we convert the OBJ key frames to a deforming mesh animation with blender shape keys [104].

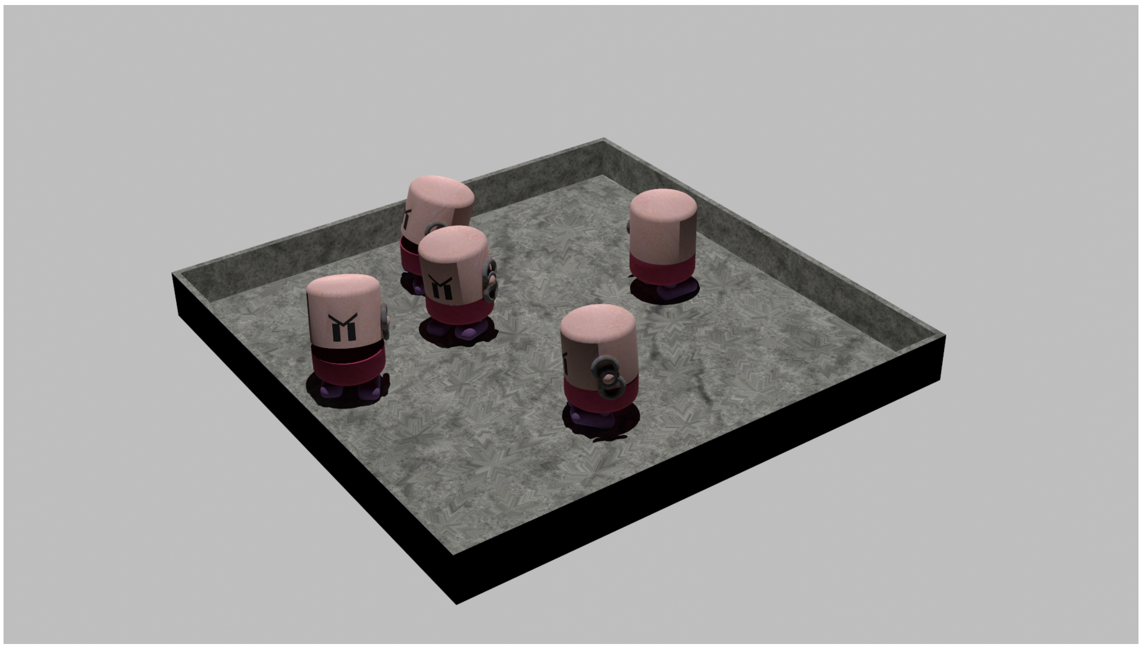

Figure 4.1. *Rendering of the Toasters scene from the Utah Animation Repository [100].*

.

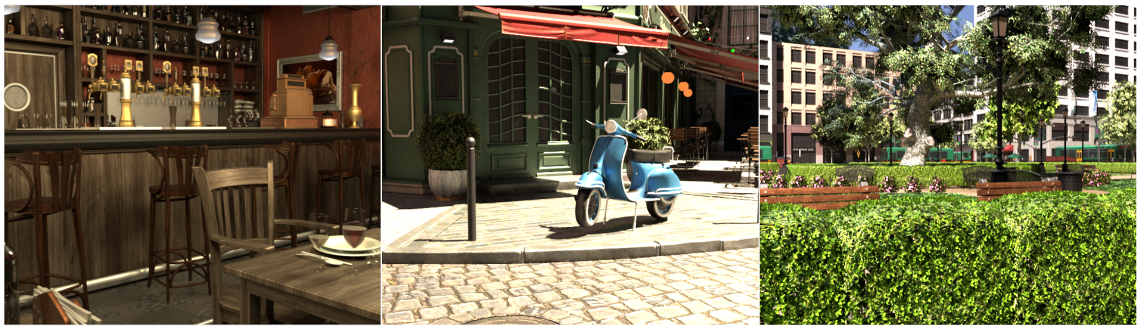

Figure 4.2. *Renderings of the three compared ORCA scenes. From left the scenes are:* BISTROINTERIOR, BISTROEXTERIOR *and* EMERALDSQUARE *[3, 67].*

4.2.2 NVidia ORCA

Nvidia Open research Content Archive (ORCA) is a professionally-created 3D assets library donated to research community. The donated sets include Amazon BISTROINTERIOR and BISTROEXTERIOR for Lumberyard game engine, NVidia EMERALDSQUARE released along with Falcor, and ZERODAY by digital creator 'Beeple' [3, 67, 105, 106, 107]. BISTRO datasets and the EMERALDSQUARE can be observed in Figure 4.2. BISTROINTERIOR and BISTROEXTERIOR were created to demonstrate new anti-aliasing and transparency features of the Amazon Lumberyard engine. EMERALDSQUARE was similarly published along with the release of the Falcor research rendering framework. All the files are in FBX file format [108], and they contain camera animations, modern textures, and modern geometry complexity. The datasets run for 60–100 seconds, and their animated cameras have 11–17 key frames described. These datasets are the current state of the art animations to be used as rendering benchmarks for temporal rendering. The only problem with the datasets is that the camera flies through the scenery very slowly and does not represent the typical use context for the algorithms. None of these datasets are released for benchmarking purposes in mind, but they are great examples of what kind of datasets are often used by the temporal research community.

To compare ORCA datasets in the same rendering context, we convert the files from the original FBX format to glTF 2.0 format using Blender. The camera animation is flipped compared to how the Falcor renderer displays it, so we apply a flip for each of the key frames, so that the camera looks in the same direction as it does with the Falcor. We also had to apply some scaling to the scenery so that one meter would represent around one unit in Blender. We applied the suggested Falcor material models in the BistroInterior dataset, namely the specific index of refraction values for glasses and liquids.

5 CAPTURING DATASET FROM CUBE 2: SAUERBRATEN

In this chapter, we describe the steps to create the dynamic benchmarking datasets. We start by describing why the glTF 2.0 was selected as the file format by comparing it to other 3D formats, and then we present a high-level view of how we utilize the glTF to define the dynamic benchmark. After that, we go in length to describe in detail the capturing workflow from Cube 2: Sauerbraten. We first start by introducing the context by taking a quick overview of the high-level game loop. We then describe offline and online capturing, then show conversion from captured intermediate format to final glTF file. Finally, we introduce the Unity VR setup for the TauEternalValleyVR dataset and how the animation is played, and how the recording is done there, followed by intermediate representation and conversion.

5.1 High Level Dataset Description in glTF 2.0

5.1.1 Comparison with Other 3D Animation Formats

The glTF 2.0 file format was released in 2015 by the Khronos Group [109]. glTF is a file format specifically created to be easily used with graphics applications. Applications can use it right away because glTF allows describing 3D primitives in OpenGL-ready binary format. glTF specification consists of all information required to represent a graphical 3D scene. There are similar formats to the selected glTF, with different tradeoffs. File formats have been used a long time, and few have placed themselves as standard. Per format, we will check its main focus and idea, what it is for, and why it has been created. There are as many file formats as 3D rendering engines, so we will review only ones that would be the appropriate alternative to glTF 2.0. These include FilmBox (FBX), Collada (DAE), and Universal Scene Description (USD).

FilmBox (FBX) file format is an industry-standard for 3D asset exchange between editors and game engines [108]. To be able to use FBX file format, a proprietary SDK must be used to handle the import and export for the binary or ASCII FBX files [110]. It supports all the essential 3D-defining elements: skins and meshes, textures, animation both rigid and skeletal, material definitions, and embedding texture images to the file, lights, and camera. However, the FBX does not support modern PBR material conventions, as it uses Blinn-Phong to describe materials' light reflectance. Its light description also lacks the current PBR standards, as it does not use physically sensible units but arbitrary light intensity values. The final and biggest downside is the fact that it does not have an official file specification. Few reverse engineering efforts have been fruitful for one of the most commonly used file formats, but overall being closed source makes it non-feasible for our use case [111].

A file format called Collada is also from the Khronos group [112]. glTF 2.0 is an upgrade and improvement over the Collada file format, so we prefer the newer one. Collada supports all the 3D defining elements, but due to ambiguous specifications, it has not been widely adopted in 3D tools [112].

Universal Scene Description (USD) is a 3D file format from Pixar [113]. In addition to the basic description of 3D properties, like meshes, lights, cameras, and textures, the USD format supports multiuser scenarios and has layers that can be stacked on top of each other. USD has been created for mainly two reasons. First, it provides a universal language for assembling, packing, and editing 3D data across different 3D authoring tools. Second, USD focuses on maximizing the collaboration on the same assets and scenes. For our use case, one unavoidable downside is the lack of rigging supports, which is left out of specification to keep the file format reasonably straightforward in the previously mentioned targets. That, and the focus on being great for artistic iteration giving additional complexity, we will not be using this file format.

There are two research renderers: Mitsuba 2 and PBRT [21, 114]. They both have their formats for scene graphs and defining materials. Their formats, however, are not supported anywhere. Even though the renderers are used a ton in graphics research and have been created mainly for it. Moreover, they lack rigging support that is a critical feature for our use case.

The open-source and new glTF 2.0 is the best selection for us. It provides all the required features, it does not have big overhead and is in an open standard format, that easily supported by any graphics program and it is already widely supported [22, 109]. Next, we will take a look at what features we need to represent the dataset.

5.1.2 Used glTF 2.0 Features and Extensions

A diagram of the hierarchy used to save the captured dataset is shown in Figure 5.1. Starting from the top, we have a glTF SCENE at root. We make the SCENE node point to the first node that we call WORLDORIGO. Like its name suggests, it is located at the origo of the scene, at coordinates $(0, 0, 0)$, and all the other nodes are placed as its child in the hierarchy. Having everything under only one nodes allows the user's of the dataset

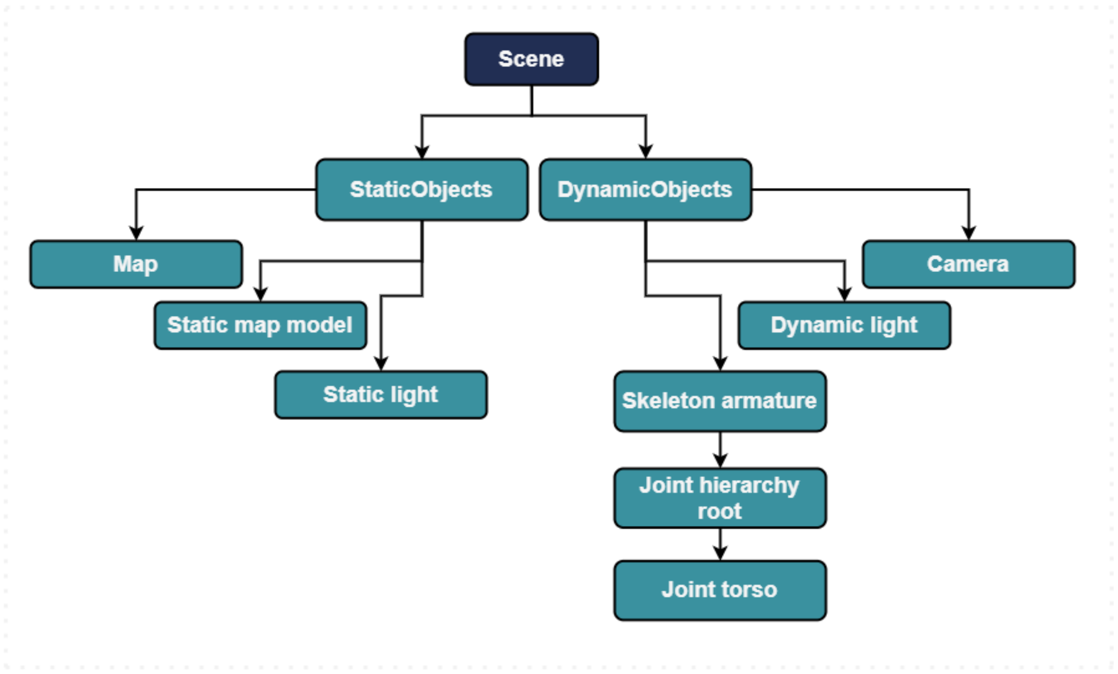

Figure 5.1. *High level figure how the GLTF 2.0 file format is used to compose the dataset. Static and dynamic objects are divided in their own branches in the tree hierarchy.*

easily to rotate and translate the scene if such need appears, and when, for example, the dataset is imported to a renderer for further inspection or editing, all of its structures are under one node.

We place two nodes under the WORLDORIGO node: STATICOBJECTS and DYNAMICOBJECTS. Straightforward naming convention tells us the purpose of these two nodes, the STATICOBJECTS node should hold all the static geometry from the scene, and the DYNAMICOBJECTS all the objects that have animation channels described for them. With this division, the clients of this dataset can quickly start without any animations if they so desire and then continue to try with rigid and skeletal animations. This division also helps with simulating an interactive ray tracing scenario, where acceleration structures must be updated due to animations [115, 116]. For example, all of the scenery under STATICOBJECTS node can be placed into a static ray tracing acceleration structure by default and dynamic to rapidly updated one. We place static models, the map, and the lights under the static branch, and dynamic camera, skeletal models, animated rigidbody models under the DYNAMICOBJECTS branch.

We use two glTF 2.0 extensions in the dataset: punctual lights and image-based lighting. Extensions are added with "extensions used" and "extension required" listings on the top level of the glTF. The punctual lights are extensions for the glTF nodes [109, 117]. Each node can point to light to animate the underlying node like we would with any other entities in the dataset. Node with the light extension must also point to corresponding light specification in lights listing in the glTF [117]. Lights can be one of three types:

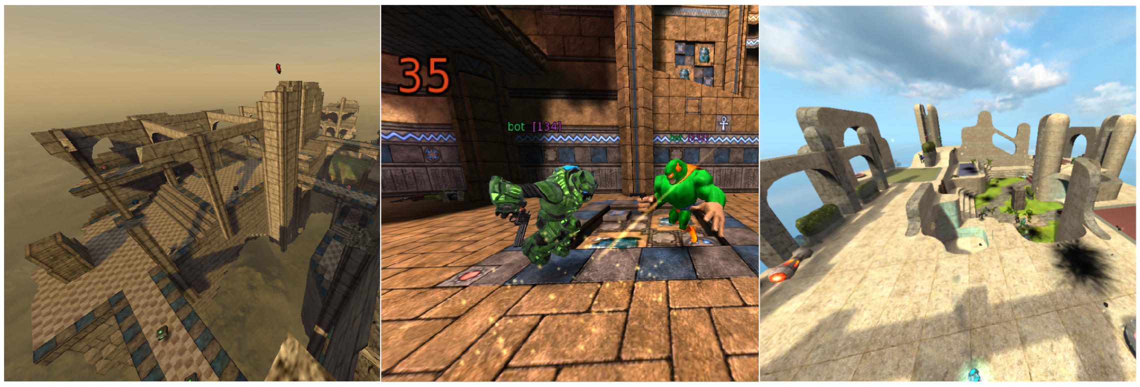

Figure 5.2. *Cube 2: Sauerbraten is an open source fast paced multiplayer arena shooter game. The screenshots captured from the game display different geometry and lighting conditions in the game.*

directional, point, or spot, from which only point lights are used with the dataset. The extension requires to specify the color of the light and the intensity of it.

The other extension we use is the image-based lighting [118]. In the extension, we represent an *environment map* that contains environmental lighting for the scene. The environment map is an approximation of the sky for a virtual world defined in a texture, and it contributes to lighting the scene [8]. Environment maps often are displayed as high dynamic range images, which encodes, also, the directional light coming from the sun [8]. We use an environment map with a permissive CC-0 license called "Kloofendal 48d Partly Cloudy" [119, 120]. The environment has outdoor scenery with a clear sky and bright sun.

5.2 Sauerbraten Rendering Loop

The temporal challenge is an intrinsic aspect of games, so to produce an animation dataset, we searched for a game that would have rapid movement for the camera and the virtual world. In addition to these, the game must be open source to make edits and apply our capturing methods. Therefore, we selected Cube 2: Sauerbraten [121]. It is a fast-paced arena first-person shooter game. Other valid games we considered were racing game SuperTuxKart, real-time strategy game 0 A.D., first-person sandbox game MineTest and slow-paced stealth game The Dark Mod [122, 123, 124, 125]. We selected the Sauerbraten, given that it would have the most significant potential of having quickly moving 3D objects and cameras.

Cube 2: Sauerbraten, depicted in Figure 5.2, is a multiplayer arena shooter game, initially developed by Wouter van Oortmerssen and Lee Salzman [126]. It diverged from the original Cube engine's source code in 2002, with a focus on developing Cube's main ideas further, namely geometrical representation and simplifying Cube's internal structures [127]. The work continues with open source contributions, and the latest version

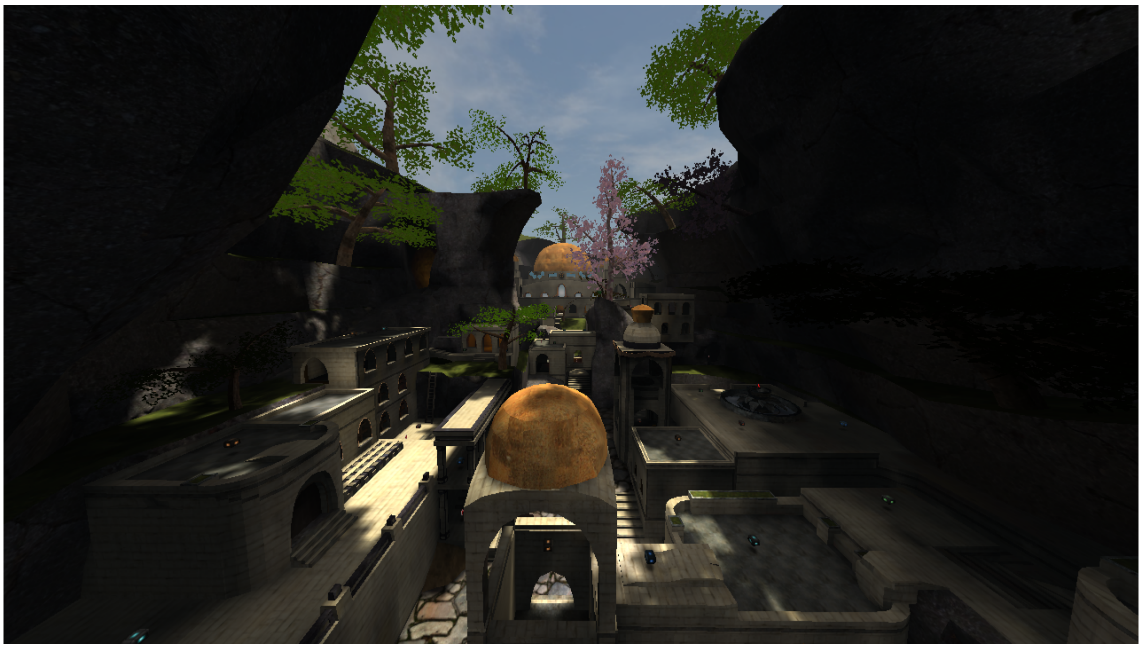

Figure 5.3. *Screen capture of the map Eternal Valley by 'Z3Ronic', created with the help of 'Hewho', 'Cooper' and 'Eihrul' to the game Cube 2: Sauerbraten [132].*

was released December 2020 [121]. The engine follows the steps of the early 2000 Id Software's game engines Quake and Doom, with a particular focus on its rendering and world creation [128, 129]. As for its mesh and material formats, it uses Doom 3's MD5 file format [130, 131].

We selected a Sauerbraten map called Eternal Valley made by user named 'Z3Ronic' for the dataset [132]. The map is a sizable outdoor scene, with the sky directly casting sunlight to the valley illuminating half of it, which can be seen in Figure 5.3. The scene has been released with Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License (CC BY-NC-SA 3.0) [120]. This license allows us to edit the map and update its textures and also permits us to publish the created dataset if we credit the creator and share it with a similar license. Other licenses in the game for models and textures are each licensed per item. Most of them have very similar licenses, varying between Creative Commons Deed / Attribution Non-commercial Share-Alike (AT-NC-SA 2.5) to Creative Commons Attribution Non-Commercial (CC-BY-NC). Unfortunately, few items in the game did not specify any license, and few had forbidding licenses, like no derivatives (CC-BY-ND) [120]. We removed items with such licenses, as they would not allow us to release this dataset as a graphics rendering benchmark.

To understand how and where to capture, we first present a simplified high-level game loop diagram of Sauerbraten in Figure 5.4. A game match in Sauerbraten starts by loading necessary resources for the selected game mode and the map. During loading, all the map entities and the player characters will be unpacked and loaded to the memory. In this loading step, the 3D models and animations will be unpacked from OBJ, MD3, and

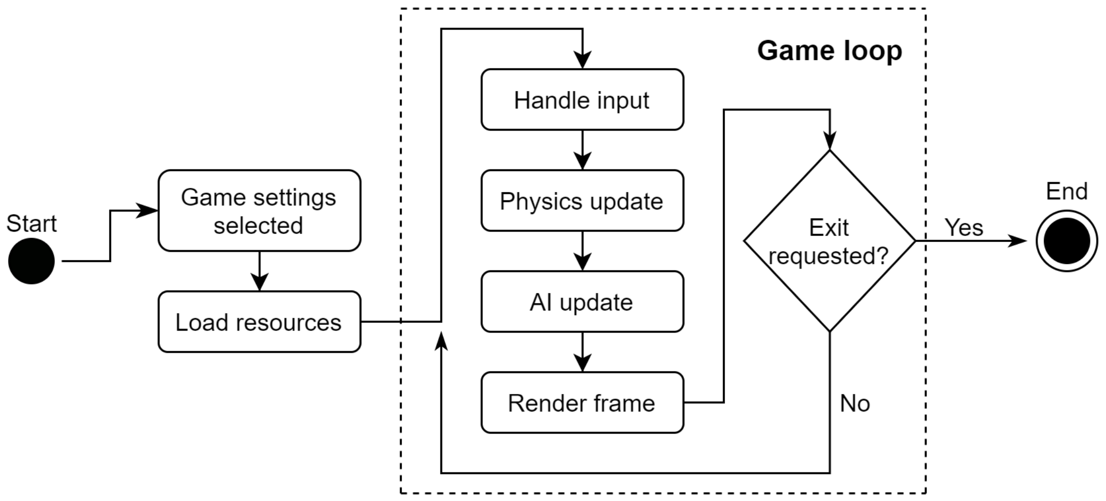

Figure 5.4. *High level diagram of the Sauerbraten's game loop.*

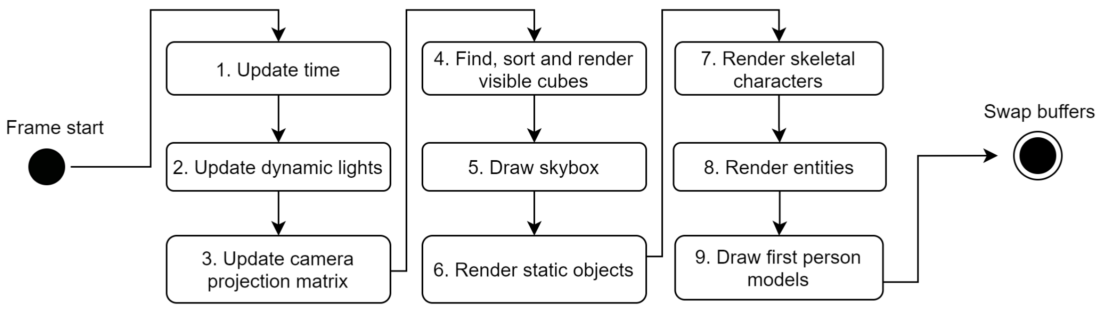
Frame start
1. Update time
2. Update dynamic lights
3. Update camera projection matrix
4. Find, sort and render visible cubes
5. Draw skybox
6. Render static objects
7. Render skeletal characters
8. Render entities
9. Draw first person models
Swap buffers

Figure 5.5. *Flow diagram of how a frame is rendered in the Cube 2: Sauerbraten.*

MD5 files to Sauerbraten's internal representation of models and animations.

When all of the necessary data is in the memory, the game loop can start. It is looped through many times per second to render a frame of the game to a screen, which we will describe shortly with the rendering loop. Each game loop iteration follows the structure game engines commonly have: first, the inputs are handled, then physics are updated and simulated, then game artificial intelligence (AI) decides the following input for them and executes it, and finally, a frame is rendered [25]. The loop stops when the game is shut down or exited.

Physics and AI updates contain essential parts for the capturing process. They update the state of all the entities in the world for the frame, and if required, their model transforms so that the rendering step can render them in the correct location [27]. These two steps may introduce new entities or remove existing ones during the gameplay.

Sauerbraten's rendering is pretty sophisticated, so in this thesis, we focus only on the parts related to capturing process [27]. We present a simplified high-level diagram of the execution flow that Sauerbraten uses to render a frame and display it on a screen in Figure 5.5. The process in Sauerbraten is executed in a single thread, making the capturing process sequential.

Next, we will advance through the Figure 5.5 step by step and present some changes to the rendering to enable successful capturing. First, the rendering starts by updating the internal time to receive the delta from the previous timestamp. Here, a correctly timestamped key frame capture is added, which will later form animations.

Next in the rendering loop, the dynamic lights are updated, and additional ones may be added. Then the player's new camera projection and all other necessary matrices view matrices are calculated. The projection is updated to the camera's new pitch and yaw values and the player's location in the scene. Then, at step 4. in Figure 5.5, the camera's frustum is used in occlusion culling and scissoring the objects not seen by the camera [76, 133]. We are interested in all of the geometry and animations happening in the scene, so we disable the geometry culling.

We disable skybox rendering that occurs on step 4 in Figure 5.5, and then, during the rendering of static objects, skeletal characters, entities, and first-person models, we capture the required model transforms. These are the key to our capturing process that we will describe in the next chapter. In addition to these steps in the diagram, there are numerous steps, like grass and water generation and rendering the Heads Up Display [HUD], that we skip, as they are not of interest in our capturing procedure. Most of these effects we disable, but they do not modify the captured outcome, only lowers the time spent in rendering.

5.3 Capture Workflow

In Cube 2: Sauerbraten, there are entirely static data, like the map and its objects, and dynamic data, like the player characters. The game loads the necessary data to random access memory at the start of the game to not disrupt the player experience during the gameplay. To create a single file for the animation, we divide the work into two steps: offline start-up and runtime captures. The workflow for capturing is depicted in Figure 5.6.

5.3.1 Offline Start Up Captures

In Cube 2: Sauerbraten, there are entirely static data, like the map and its objects, and dynamic data, like the player characters. The game loads the necessary data to random access memory at the start of the game to not disrupt the player experience during the gameplay. We perform the offline capture just before the interactive game loop starts. Capturing is performed during the loading resources step shown in Figure 5.4. At this point, Sauerbraten models and Sauerbraten animations are formed and unpacked from MD5Mesh and MD5Anim files. We capture this Sauerbratens internal model, as presented in Figure 5.6. Inspired by standard human-readable file formats, like OBJ [134],

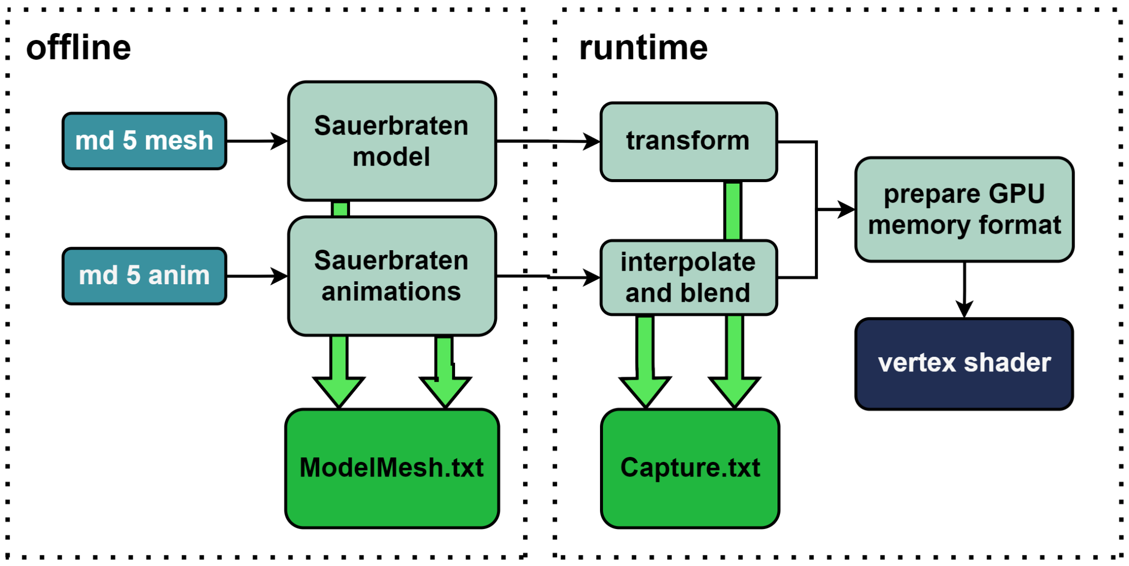

Figure 5.6. *Diagram shows how MD5Mesh and MD5Anim files together are parsed to the Cube 2: Sauerbraten's internal representation. We do the offline capture here for each used model. Instances of the models move around during runtime, and their pose change accordingly. Final vertex skinning happens in the vertex shaders. We capture the pose and the model dual quaternion just before that, after all the game related changes have been applied to the animations.*

we record the offline resources to an intermediate format described in Table 5.1.

For each of the captured properties, we define a parsing header and the format for the data. The headers start with a unique property identifier, like p in Table 5.1, which identifies that the following data in the string line holds a vertex position. The other header keys are the mesh identifier, $meshID$, which describes which of the meshes this model belongs. Character models commonly have multiple meshes in the Sauerbraten [131]. However, the armatures expressed with bind and inverse bind poses are shared with all the meshes in the model.

Model properties' data formats contain the data in the described format in Table 5.1. All of the properties follow the standard practice used in rasterization that the data is connected to a single vertex so that it can be efficiently processed with a vertex shader [8]. So the normal is the surfaces normal at the point of the vertex, and joints show which joints influence this vertex.

In Sauerbraten's internal models, each vertex will have at least one joint influencing it and a maximum of four joints. Therefore, there may not be all of the elements present for each joint and weight. In such case, we show the data as symbol *?* in Table 5.1.

Model's armature is described with a resting bind pose b, and an inverse bind pose x. We retrieve the bone hierarchy, as we add an identifier of the parent bone with $jointID$. In addition, there is a parent index for each of the joints, which can be used to link the joints

Table 5.1. *Captured properties of a single model to ModelMesh.txt files during the offline capturing procedure.*

model property	parsing header	data format
vertex position	p mesh ID	x : y : z
normal	n mesh ID	x : y : z
triangle index	i mesh ID	i : j : k
texture coordinates	t mesh ID	u : v
joint	j mesh ID	x : y? : z? : w?
weight	w mesh ID	x : y? : z? : w?
texture map	a mesh ID	texture name
texture mask map	g mesh ID	texture name
map	u map name	x : y : z
camera settings	c	aspect ratio : yFov : zNear : zFar
bind pose	b	joint ID joint name parent index rx : ry : rz : rw ; dx : dy : dz : dw per joint
inverse bind pose	b	joint ID joint name parent index rx : ry : rz : rw ; dx : dy : dz : dw per joint

Table 5.2. *Captured properties saved to Capture.txt of the game state during runtime.*

captured property	parsing header	data format
frame time	f	frame number frame time
static map model	p capturable ID model path	rx : ry : rz : rw dx : dy : dz : dw
camera transform	t capturable ID	x : y : z pitch : yaw : roll
light	l capturable ID light type is dynamic? model type	x : y : z R : G : B
model transform	m capturable ID model path	rx : ry : rz : rw dx : dy : dz : dw
animated joints	j capturable ID model path	rx : ry : rz : rw dx : dy : dz : dw per joint

in a hierarchy. With bind pose, each dual quaternion of each joint holds the resting pose in local space format. Inverse bind pose is the opposite joint transform, moving joints from animation space pose to local joint space.

The selected map is recorded only by its name and the position offset its origin has from Sauerbraten world's origin coordinates. The name can be later used to retrieve the model for the map.

The final captured properties are the textures. Each model has corresponding texture maps and texture mask maps. The map has the base color, and the mask map has the normal map and specular glossiness information. Data that is captured is only the name of the used image file, then retrieved later.

5.3.2 Runtime Captures

Physics update and AI update depicted in the game loop Figure 5.4 updates each frame's state. This state holds transforms, blended skeletal animations, and other entity properties. We present a similar capturing strategy shown with the offline capturing in Table 5.2.

Each property is identified with a parsing header in the final Capture.txt. We call the recorded entity instances *Capturables*, and add a unique identifier $capturableID$ for them with. Instances will appear and disappear during the rendering process, so we generate a new running identifier for them on the fly. We bind the info declared with data format as an animation key frame for the given Capturable instance.

At the start of a new frame, at step 1. in Figure 5.5, we capture a new moment in time with a frame time. The captured moment is the time that will be applied as a key frame time to all of the new animation transitions that happen during rendering this frame. Transitions will end up in the intermediate file in chronological order simplifying the parsing process. However, recording key frame times means that the capturing will be dependant on the game rendering frequency. To ensure the recorded animations stay stable, we set a fixed rendering rate to 60 frames per second.

Cube 2: Sauerbraten entities are specified to 5 different properties: static models, camera, lights, dynamic models, and skeletal models. Mesh-related properties have model paths declared in their header. The path can be later used to point to the correct offline recorded ModelMesh.txt when converting to the final animation format.

Before capturing most of the properties, Figure 5.6 shows how Sauerbraten must first update their model transform, or with skeletal armatures interpolate and blend between animations. The transforms are represented as translations and rotations or as a dual quaternion.

Moving player characters are described with both a model transform and a joint hierarchy, and they share $capturableID$. The model transform places the model origin to the world coordinates, and the armature transforms the joints in model space. Sauerbraten uses a standard skinning procedure with the vertex shader like shown in Appendix A.1, so it must prepare the animation joint matrix stack for each of the skeletons in the scene [27]. We capture the joints' dual quaternions after being computed to their animated pose in the model space, but just before they are converted to a local joint space with their parent joint's inverse bind pose [8, 25]. With such capturing strategy, we reduce the steps in the conversion workflow described later.

Final Sauerbraten specific Capturables are the lights in the scene. They have a color, a light type, a model type, and they can be either static or dynamic. We record the color with RGB values the lights have internally in Sauerbraten. The light type declares whether the light is a point light, a directional light, or a spotlight. However, we only use point lights and other types we filter out during the capture parsing process. The dynamic or static flag tells us whether or not the light will have model transforms. The final item in the light's parsing header is the model type. The type is a Sauerbraten specific identifier that tells us whether the light entity is a flash, a bouncer, or a rocket [131]. We use different conversion methods for each of these types to make them look very similar to how it appears in the

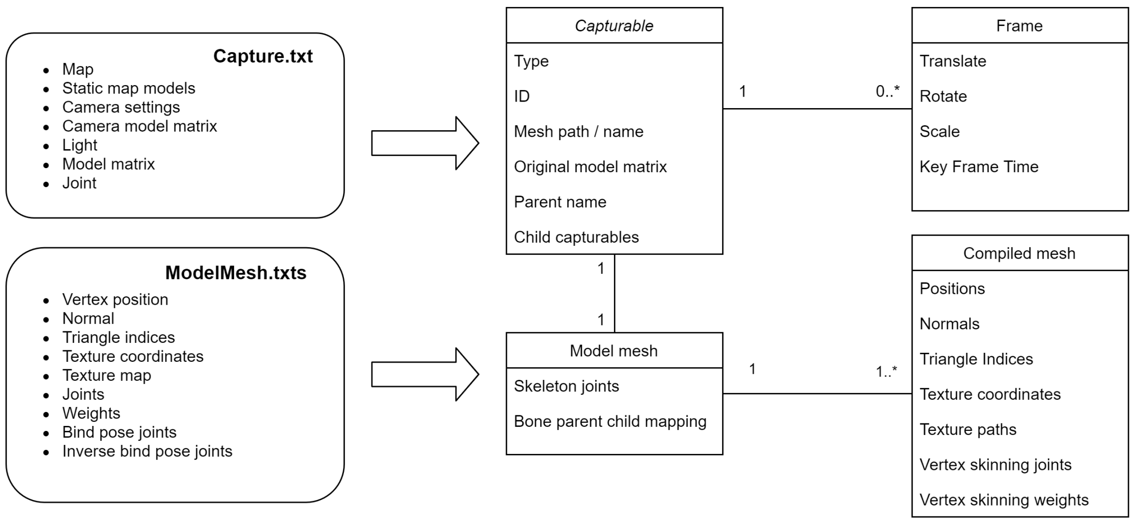

Figure 5.7. *High-level class diagram of parsing the capture data to Capturables and ModelMeshes.*

game, even when the dataset is path traced.

5.4 Conversion Workflow

Next in the pipeline is to start building the glTF file. With the help of the parsing headers shown in Tables 5.1 and 5.2, we parse the intermediate capture files to classes depicted in Figure 5.7. These classes are Capturable, Frame, ModelMesh, and CompiledMesh.

Capturables contain the recorded entity instance information for each recorded models. Captures may have Frames, which describe animation properties per key frame basis. If Capturable does not have Frame, the Capturable is marked as static. Capturables will always have ModelMesh connected to it. ModelMeshes are similar to the models in Cube 2: Sauerbraten, describing singular skeletal or rigidbody model [131]. Each ModelMesh will have one or more of CompiledMeshes that describe the models' 3D primitives.

Each compiled mesh can be put to the glTF file. They are placed in Meshes in the glTF specification, with all the primitives described as corresponding buffers. For example, the vertices from the compiled mesh can be placed under glTF Mesh components' *Position* attribute, which points to OpenGL buffer [109]. All the other primitives have a similar kind of place in the glTF's mesh. The meshes need to be only defined once, and then multiple Capturables may point to the same mesh and its primitives.

The model transforms used to define the placements of these Capturables in the scene. We use captured Frames to create animation channels for the translation, the rotation, and the scaling that happens during the animation [109]. Recorded key frames are applied as-is for the animation. When the object is static, only the known placement is used as the node's transform matrix.

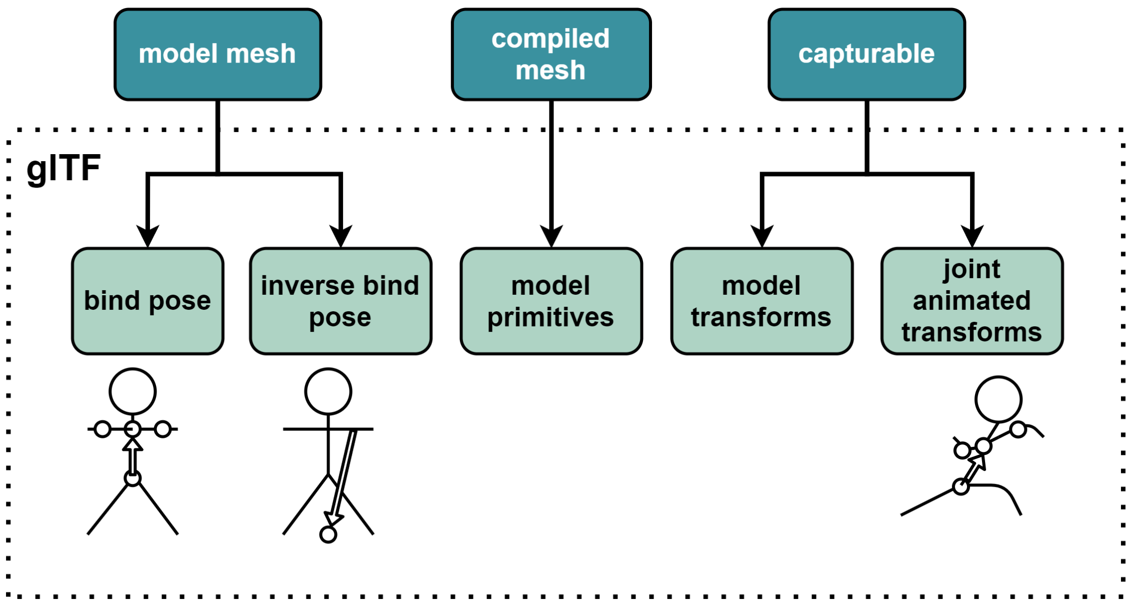

Figure 5.8. *Diagram shows how the ModelMesh, CompiledMesh and Capturable are contributing to different parts of the final glTF dataset. glTF has the skeletal data in slightly different format than that of the MD5Mesh and MD5Anim or the Sauerbraten's internal model, so they must be rotated and converted correctly.*

Because there are appearing and disappearing objects in the captured animations, we use scaling to determine when the Capturable should be shown in the animation and when not. We set the scale of the object to 0 for those key frames that the Capturable should be invisible, and when it reappears, we return their correct scaling values to normal.

We convert dual quaternions to translations and rotations with the recorded skeletal armatures and decompose the skeletal parent-child hierarchy. In the glTF specification, the bind pose must be created with glTF's nodes, placing them in a similar child-parent hierarchy. We place the joint orientations as the resting bind pose from ModelMesh like depicted in Figure 5.8. Animations are simply applying the same kind of animations that was done with the model matrices, but now we rotate and translate these individual bones [109].

The camera is simple to declare with glTF: we only need to place the parameters required for a pinhole projection matrix. These are the information we have successfully captured with aspect ratio, yFov, and xFov. Its animation follows the same procedure as any other animation in glTF, in which the underlying node is animated.

The final things are the lights and like previously mentioned in Chapter 5.1.2, they are simple extensions on top of glTF nodes. We utilize the earlier techniques to place the node's wrong spot in the world, and if the lights are dynamic, we add corresponding translation and rotation animation channels for them. Furthermore, if the light appears and disappears during the animation, we set their scale to zero, indicating that the light should not be used in rendering on this frame. We recorded three types of lights during the gameplay: bouncers, flashes, and rockets. For bouncers, we add and offset six lights from its recorded position to around the bouncer and use the recorded color and values weighted so that the rendered image looks approximately the same as it does in the game. Rockets do not need any edits, and flashes stay only for a short amount of time. We amplify the intensity values of the flashes and stay few frames longer to more closely representing the effect that happens in the game.

5.5 Capturing Virtual Reality

The first-person camera does not have roll rotation, as we showed in the camera captures process, where we capture only pitch and yaw. For this reason, we also want to produce a dataset that would have a realistic VR setting in terms of temporal rendering. We load a recorded glTF to Unity, then we setup simple VR movement with the Oculus 2 Unity integration sample set, then use the Oculus Quest 2 VR set to view a previously recorded gameplay match and finally capture a camera recording [135, 136].

Unity is a 3D game engine by Unity Technologies [135]. Unity has been used a lot recently, for example, in games Fall Guys, Valorant, and also in different industries, like in automotive by Audi and Toyota [137]. It also has excellent support for AR and VR. For example, games BeatSaber and PokemonGo have been created with it [137]. We setup a simple new Unity scene and add the animation file to it with glTF Unity tool package UnityGLTF by the Khronos Group [138].

Oculus Quest 2 is a VR headset [136]. The headset runs Android application packages without high-end host PC GPU, and it tracks the user headset motion with its cameras. A VR SDK provided by Oculus comes with an example program that perfectly suits our use case. The scene is configured to use both VR movement styles: walking and teleporting [136]. We load our animation to the scene, add collisions to the static meshes, scale the world correctly and configure the walking speed. We also add looping to the animator. With this setup, we can walk around and see the game match as an animation running like it was recorded from the game.

Finally, we add a script to the created Unity that records each animation run to a Capture.txt file, with only camera matrices for each of the frames. With the recordings, we use the conversion workflow described earlier to convert all the animations to camera runs and combine that with the previous recording we loaded to Unity. Now we have captured and created EternalValleyVR glTF file that represents well how the camera moves when used with VR applications.

6 RESULTS

In this chapter, we compare the produced datasets against the previously published work discussed in Chapter 4.2. First, we present the details and properties of the animation datasets. Then, we perform measurements on the camera's spatial movement during the animation. Finally, we reproject each frame's pixels to the previous frame and compare how much the frame should be discarded as invalid history. We discard the pixel if there are significant enough changes in the expected feature buffers. The buffers we focus on for these comparisons, are often used to guide in temporal shading reuse algorithms and post processing effects, like the temporal anti-aliasing. Namely, we compare the changes in the virtual camera's view frustum, pixel's world position, shading normal, direct and indirect radiance.

6.1 Dataset Properties

Properties of each dataset are presented in Table 6.1. Proposed datasets are the first to use the file format glTF 2.0, while others use the file formats FBX and Wavefront OBJ, like discussed in Chapter 4.2. The face count gives some idea how complex each of the scenes are ORCA scenes BISTROINTERIOR and BISTROEXTERIOR being the most complex and Utah TOASTERS the least. Utah Animation repository is the only one compared here that does not have an animated camera, and its geometry is animated with morph targets.

Next, the texture details for each dataset can be seen in Table 6.2. On the other hand,

Table 6.1. *Datasets' properties.*

	file format	file(s) size	faces	animated camera	morph targets
Eternal Valley FPS	gLTF	1429 MiB	240 584	Yes	No
Eternal Valley VR	gLTF	1564 MiB	292 475	Yes	No
Toasters	wavefront obj	163 MiB	11 141	No	Yes
Bistro Interior	fbx	559 MiB	1 248 093	Yes	No
Bistro Exterior	fbx	1098 MiB	2 829 226	Yes	No
Emerald Square	fbx	2459 MiB	2 691 019	Yes	No

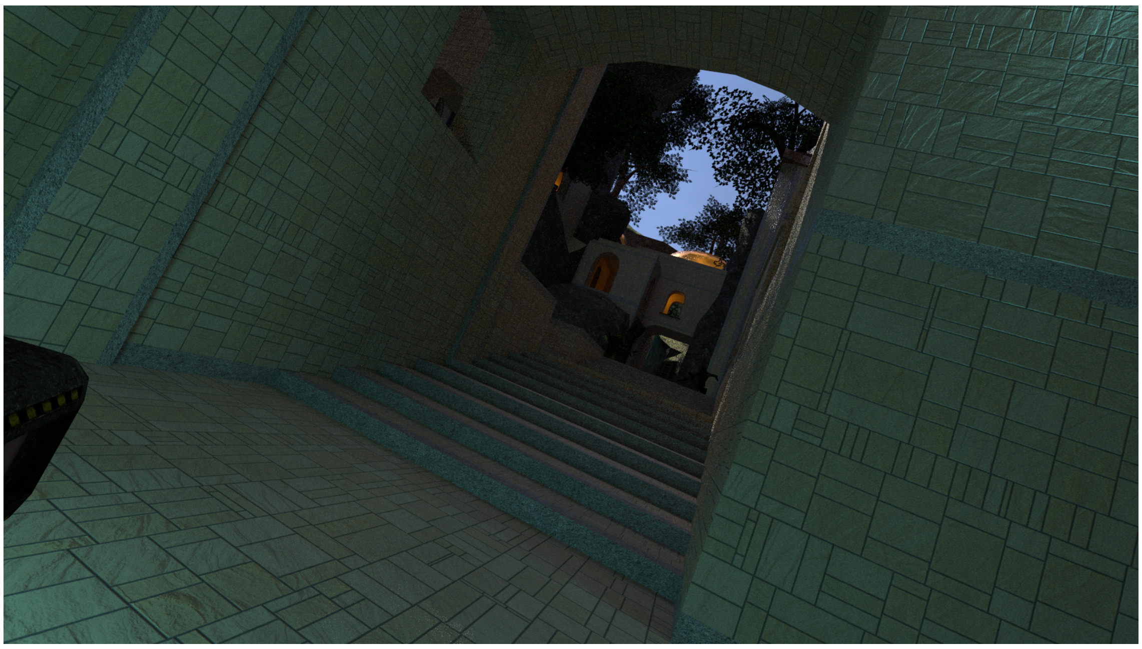

Figure 6.1. *Rendered image of the* ETERNALVALLEYVR *dataset. The roll rotation is apparent throughout the animation.*

Table 6.2. *Datasets' texture details.*

	material count	texture count	texture size	material workflow
Eternal Valley FPS	89	314	1406 MiB	PBR roughness metallic
Eternal Valley VR	89	314	1406 MiB	PBR roughness metallic
Toasters	7	6	0,62 MiB	Phong
Bistro Interior	74	212	512 MiB	PBR roughness metallic
Bistro Exterior	132	417	984 MiB	PBR roughness metallic
Emerald Square	220	701	2293 MiB	PBR roughness metallic

material definitions reveal how densely the geometry reacts to the lighting condition and how realistic the objects look when rendered. We note that Utah Animation has the least complexity in its textures and uses Blinn-Phong shading with just simple diffuse materials, whereas the rest of the datasets are using more modern physically-based materials. The outdoor scenes BISTROEXTERIOR and EMERALDSQUARE have considerably higher count of materials than others sets, with EMERALDSQUARE having over 700 different textures used in 220 materials. ETERNALVALLEY datasets sit in the middle of the compared sets, with 89 materials that use 314 textures.

Finally, the details of the animation of each dataset have been noted in Table 6.3. Animation sizes vary from around 200 frames to over 2400, ranging in 24 frames per second from 8 seconds to 100 seconds long. The proposed dataset sits in the middle, with around 30 seconds of animation. The Utah animation TOASTERS have the animation run for the whole duration, but being a morphed target animation, it lacks the object identifiers, mak-

Table 6.3. *Datasets' animation details.*

	animation frames	static objects	dynamic rigid objects	armatures	static point lights	dynamic point lights
Eternal Valley FPS	760	228	3100	34	46	1266
Eternal Valley VR	760	228	4333	32	46	1439
Toasters	248	-	-	-	-	-
Bistro Interior	1433	1414	0	0	4	0
Bistro Exterior	2404	1281	15	0	1	0
Emerald Square	1541	1030	0	0	2	0

Table 6.4. *Change in camera's position.*

	mean	variance	max
Eternal Valley FPS	4.711	1.741	10.855
Eternal Valley VR	0.078	0.252	8.859
Toasters	0	0	0
Bistro Interior	0.013	0.000	0.018
Bistro Exterior	0.055	0.001	0.143
Emerald Square	0.154	0.004	0.248

ing it non-trivial to utilize acceleration structures in the recent ray tracing APIs [101].

Like with the face count in Table 6.1, the object count adds to the information on how complex the given scene is. There are remarkably fewer static objects in the proposed datasets compared to the ORCA ones. The EternalValley sets have more dynamic rigidbody objects and dynamic lights than the compared datasets. The count of animated objects continues the trend of the dataset following closer to an actual setting of an interactive scenario where the objects and lights movement is animated by the game automatically, and not by an animator by hand. Lastly, the new proposed datasets are the only sets with animated armatures.

6.2 Camera Movement

In dynamic scenes, in order for the renderer to have a temporal challenge in the rendering process, either the camera or the scene must change their transform. To validate the amount of challenge the proposed dataset opposes on a rendering client, we start by taking a look at the camera's change in position and rotation during the animation.

We calculate the distance the camera moves each frame using the Eq. (3.1). The calculated mean, variance, and the maximum value of the movement can be seen in Table 6.4.

Table 6.5. *Change in camera's rotation.*

	mean	variance	max
Eternal Valley FPS	1.931	4.526	13.665
Eternal Valley VR	1.857	8.142	33.092
Toasters	0	0	0
Bistro Interior Wine	0.136	0.012	0.322
Bistro Exterior	0.134	0.013	0.472
Emerald Square	0.258	0.023	0.554

The amount of camera transforms around the scene varies wildly between the datasets. One noticeable aspect of the movement in the proposed dataset are the continuous small changes in its motion compared to other datasets. Continuous change is shown in the more considerable variance in the ETERNALVALLEY datasets. Other datasets keep the constant value for a while, then jump abruptly to a new one. This shines some light on the main difference between the proposed dataset and the previous work: our dataset's animation key frames have been recorded during the gameplay with high frequency, whereas the previous work has the key frames placed by an animator, letting the camera fly between marked points linearly.

Similarly to the position, we calculate the camera's intrinsic geodesic distance between the two orientation quaternions for each frame in the animation using Eq. (2.4), take their average, variance, and maximum value, and present them in Table 6.5.

The same variance difference can be noticed in the datasets' rotations in Table 6.5. Furthermore, shown max values are much more significant in the proposed sets. The maximum value can be considered the worst-case scenario: how significant a change will be between two different frames. Proposed datasets' maximum deltas are about $20\times$ compared to their average changes show that there are some challenging situations for temporal algorithms, whereas the compared previous works' max values are only $3\times$ higher than the average values. Regardless of the mean of the positional change being only slightly higher than the movement in BISTROINTERIOR and BISTROEXTERIOR, and more minor in EMERALDSQUARE, the diminishingly low variance in these scenes shows how predictable the movement is.

While the position is dependent on the scale of the world, giving only a little information of the temporal challenge it proposes on the renderer, the camera's rotation is not. The camera can rotate a maximum of 180 degrees per frame, so it is a better-suited metric when comparing datasets' camera activity. Rotation reveals new scenery previously not seen by the camera, so the bigger the change, the less can be reused from the previous frame. Proposed datasets have spikes throughout the animation that have significantly more significant angular change than the previous work. This is shown as variance in

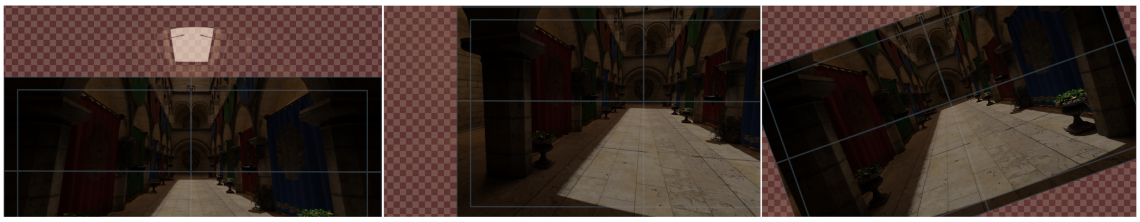

Figure 6.2. *Rendering of Sponza scene with camera turned 25 degrees from left with pitch rotation upwards, center image yaw rotation to left and on the right roll rotation [2]. On red are the frustum discard areas that does not have without any valid temporal history data. Quick movements especially in yaw direction rapidly invalidates all the available history, where as roll rotation can invalidate only the corners.*

Table 6.6. *Change in camera's rotation per axis.*

	pitch mean (degrees / frame)	pitch variance (degrees / frame)	yaw mean (degrees / frame)	yaw variance (degrees / frame)	roll mean (degrees / frame)	roll variance (degrees / frame)
Eternal Valley FPS	1.088	2.450	3.454	17.468	0.004	0.000
Eternal Valley VR	0.939	1.298	3.365	33.086	0.557	0.405
Toasters	0	0	0	0	0	0
Bistro Interior	0.053	0.004	0.266	0.046	0.002	0.000
Bistro Exterior	0.014	0.000	0.265	0.051	0.004	0.000
Emerald Square	0.044	0.002	0.511	0.095	0.000	0.000

rotation in Table 6.5.

The type of the rotation also matters for the temporal rendering challenge: the same amount of yaw rotation requires more frustum rendering than roll rotation like seen in Figure 6.2, but the roll rotation might be more challenging to temporally accumulate than simple horizontal or vertical rotation [52].

We calculate the change in rotation around each of the camera's three-axis, pitch, yaw, and roll, individually using Euler angles with Eq. (3.2), and compare their means and variances in Table 6.6.

The proposed datasets have a more considerable change in all angles during the animation than any other set. Moreover, compared to the others, the dataset ETERNALVALLEYVR seems to be the only dataset with considerable roll rotation, like seen in the rendered image in Figure 6.1. This can be explained by the fact that it was captured from a virtual reality setup, in which the user constantly sways their head slightly during the recording. The most active compared animation EMERALDSQUARE has the highest average in yaw rotation of the previous work, but it is still $6\times$ smaller than the proposed dataset. Furthermore, the variance of that set is $649\times$ more minor in pitch rotation and $348\times$ smaller in yaw rotation.

Time-series graphs of these rotational changes can be found in Appendix A.2.

6.3 Discard Percentage

The final aspect of the datasets we are interested in is the discard percentages. To accelerate the rendering of an image, the renderer can guide its efforts to low confidence areas. Temporal methods accumulate pixel's history information but cannot do so when a new frame's pixel reveals new geometry or the history becomes invalid as the settings for the lighting of the pixel have changed. Our experimental discard method follows essentially the idea behind temporal accumulation methods: using the information in previous frames feature buffers to validate how usable the history information is. The discarding process starts by first reprojecting the pixel to the previous frame, then use the camera's rotational movement to discard frustum edges fresh for the given frame, and finally, continue to compare feature buffers with each pixel.

We render datasets' animation with Blender's path tracer Cycles and extract feature buffers, namely world-space positions and normals, direct and indirect radiance, separately [139]. Blender's Cycles is a path tracing renderer with a plethora of supported features. Notably, the ability to render skeletal armatures was one key reason to select it as the renderer for the comparisons. Other renderers designed for research purposes, like PBRT or Mitsuba 2, lack this feature [21, 114]. All of the feature buffers are rendered with resolution 1920×1080 pixels, and for the indirect buffer, the pixels are sampled 1024 times with 12 next event estimation light bounces. We selected these configurations to have reasonable rendering time and good enough sample count so that the indirect buffer has time to converge enough so that the noise is mostly mitigated. Rendering in higher resolution, using a bigger sample size and a longer sample path would result in higher precision discard percentages, but selected rendering configuration allows us to compare the datasets.

For each inspected feature buffer, we take the discard percentage for each scene with the corresponding discard functions presented in Chapter 3. Finally, we sum the per frame percentages with Eq. (3.4) and show the mean of the percentages in Table 6.7.

ETERNALVALLEYFPS has the biggest percentage of discarded pixels due to frustum disocclusion, and ETERNALVALLEYVR coming right behind. Both datasets have a much more significant frustum discard percentage compared to the ORCA sets. This lines up with the previously recognized change in the motion of the camera. The same trend continues with all of the discard properties, ETERNALVALLEYFPS having the highest dis-

Table 6.7. The percentage of discarded pixel averaged over the length of the animation.

	frustum %	world position %	shading normal %	direct radiance %	indirect radiance %
Eternal Valley FPS	29.085	33.521	47.671	8.583	6.403
Eternal Valley VR	19.255	29.956	23.455	5.824	8.077
Toasters	0.0	0.214	1.073	0.973	0.432
Bistro Interior	0.140	2.156	0.962	2.769	0.718
Bistro Exterior	1.192	4.502	2.207	0.706	0.401
Emerald Square	1.781	11.416	17.461	4.983	0.432

card rate, and ETERNALVALLEYVR the second most. The dataset EMERALDSQUARE does have quite a sizeable discard percentage with world position and shading normal compared to the other ORCA sets. This most likely is explained due to the amount of vegetation the scene has, as it is filled with bushes and trees filled with individual leaves.

Proposed datasets show also higher discard due to changing lighting conditions: ETERNALVALLEYVR seems to have the greatest change in the indirect radiance of all the scenes. ETERNALVALLEYFPS on the other hand, has the most change with directional lighting conditions. BISTROINTERIOR and EMERALDSQUARE do have quite a bit of change in their direct radiance, but they are still much lower than the proposed sets.

7 CONCLUSION

In this thesis, we sought to determine the current status of temporal rendering and benchmarks available to challenge the methods and produce applicable dynamic benchmarks. There are no publicly available datasets that can bring forward issues temporal reuse methods have. Issues include ghosting, blurring, and aliasing in rasterization, judder in VR, and light update lag in real-time ray and path tracing.

This thesis produces two datasets, ETERNALVALLEYVR and ETERNALVALLEYFPS, that contain an excellent basis to benchmark temporal rendering. While the proposed scenes have fewer surface faces than most of the compared ORCA datasets, they have a higher amount of dynamically moving objects and are the only sets with skeletal animations, as shown in Chapter 6.1. We show that these produced datasets exceed previously released benchmarks in their camera's motion, in both cameras positional change, and in their rotation in the Chapter 6.2. The camera's animated transform determines how challenging the temporal reuse cases are, and we exceed the previously released datasets. The amount of challenge is confirmed even further in the discard percentages shown in Table 6.7, where we show that the features used as input to most of the temporal reuse algorithms change more rapidly on our animations. In summary, we confirm that our datasets have the most considerable potential of being used as benchmarks for temporal reuse algorithms.

In addition to the two datasets, we introduced a framework that can capture the animations out of an interactive system, and we verified that it works by producing our datasets.

Interesting future work would be to generalize the framework further and use it in different interactive systems. The comparison metrics could also be extended to cover emissive triangles more depth, which are often commonplace in path tracing settings. A straightforward extension would also be to separate the direct and indirect radiance to diffuse, glossy, transmissive, and volumetric passes. The division would allow finding a benchmark that excels in challenging temporal methods in cases where the data flows from reflections indirectly, and more robust motion vectors must be used.

REFERENCES

[1] Tamstorf, R. and Pritchett, H. The challenges of releasing the Moana Island Scene. (2019). DOI: 10.2312/sr.20191223.

[2] McGuire, M. *Computer Graphics Archive*. 2017. URL: https://casual-effects.com/data (visited on 03/18/2021).

[3] Nicholas Hull, K. A. and Benty, N. *NVIDIA Emerald Square, Open Research Content Archive (ORCA)*. July 2017. URL: http://developer.nvidia.com/orca/nvidia-emerald-square.

[4] Abrash, M. *Why virtual reality is so hard (and where it might be going)*. 2013. URL: https://media.steampowered.com/apps/valve/2013/MAbrashGDC2013.pdf (visited on 03/18/2021).

[5] Richter, S. R., Hayder, Z. and Koltun, V. Playing for Benchmarks. *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*. 2017, 2232–2241. DOI: 10.1109/ICCV.2017.243. URL: https://doi.org/10.1109/ICCV.2017.243.

[6] Richter, S. R., Vineet, V., Roth, S. and Koltun, V. Playing for Data: Ground Truth from Computer Games. *European Conference on Computer Vision (ECCV)*. Ed. by B. Leibe, J. Matas, N. Sebe and M. Welling. Vol. 9906. LNCS. Springer International Publishing, 2016, 102–118. DOI: arXiv:1608.02192.

[7] Butler, D. J., Wulff, J., Stanley, G. B. and Black, M. J. A naturalistic open source movie for optical flow evaluation. *European Conf. on Computer Vision (ECCV)*. Ed. by A. Fitzgibbon et al. (Eds.) Part IV, LNCS 7577. Springer-Verlag, Oct. 2012, 611–625. DOI: 10.1007/978-3-642-33783-3_44.

[8] Akenine-Möller, T., Haines, E. and Hoffman, N. *Real-time rendering*. Crc Press, 2019.

[9] The Khronos Group, Inc. *OpenGL 4.5 Reference Pages*. 2021. URL: https://www.khronos.org/registry/OpenGL-Refpages/gl4/ (visited on 03/16/2021).

[10] The Khronos Group, Inc. *Vulkan*. 2021. URL: https://www.khronos.org/vulkan/ (visited on 03/16/2021).

[11] Microsoft Corporation. *Direct3D 12 reference*. 2019. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-reference (visited on 03/16/2021).

[12] Kajiya, J. T. The Rendering Equation. *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902.

[13] Immel, D. S., Cohen, M. F. and Greenberg, D. P. A Radiosity Method for Non-Diffuse Environments. *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), 133–142. ISSN: 0097-8930. DOI: 10.1145/15886.15901.

[14] Karis, B. and Games, E. Real shading in unreal engine 4. *Proc. Physically Based Shading Theory Practice* 4 (2013), 3.

[15] Nicodemus, F. E. Directional Reflectance and Emissivity of an Opaque Surface. *Appl. Opt.* 4.7 (July 1965), 767–775. DOI: 10.1364/AO.4.000767.

[16] Torrance, K. E. and Sparrow, E. M. Theory for off-specular reflection from roughened surfaces. *Josa* 57.9 (1967), 1105–1114. DOI: 10.5555/136913.136924.

[17] Cook, R. L. and Torrance, K. E. A reflectance model for computer graphics. *ACM Transactions on Graphics (ToG)* 1.1 (1982), 7–24. DOI: 10.1145/357290.357293.

[18] Trowbridge, T. and Reitz, K. P. Average irregularity representation of a rough surface for ray reflection. *JOSA* 65.5 (1975), 531–536. DOI: 10.1364/JOSA.65.000531.

[19] Walter, B., Marschner, S. R., Li, H. and Torrance, K. E. Microfacet Models for Refraction through Rough Surfaces. *Rendering techniques* 2007 (2007), 18th. DOI: 10.5555/2383847.2383874.

[20] Phong, B. T. Illumination for Computer Generated Pictures. *Commun. ACM* 18.6 (June 1975), 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839.

[21] Pharr, M., Jakob, W. and Humphreys, G. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[22] Blender Foundation. *Blender*. 2021. URL: https://www.blender.org/about/ (visited on 03/01/2021).

[23] Veach, E. and Guibas, L. J. Optimally Combining Sampling Techniques for Monte Carlo Rendering. *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '95. New York, NY, USA: Association for Computing Machinery, 1995, 419–428. ISBN: 0897917014. DOI: 10.1145/218380.218498.

[24] Ahn, S. H. *OpenGL Transformation*. 2019. URL: http://www.songho.ca/opengl/gl_transform.html (visited on 03/16/2021).

[25] Gregory, J. *Game engine architecture*. crc Press, 2018.

[26] Hoag, D. Apollo guidance and navigation: Considerations of apollo imu gimbal lock. *Canbridge: MIT Instrumentation Laboratory* (1963), 1–64.

[27] *Cube 2: Sauerbraten (game engine & FPS)*. 2021. URL: https://sourceforge.net/projects/sauerbraten/ (visited on 03/15/2021).

[28] Ed., S. W. R. H. L. P. F. H. M. R. S. and Corr. M., D. H. or. II. On quaternions; or on a new system of imaginaries in algebra. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 25.163 (1844), 10–13. DOI: 10.1080/14786444408644923.

[29] Shoemake, K. Animating Rotation with Quaternion Curves. *SIGGRAPH Comput. Graph.* 19.3 (July 1985), 245–254. ISSN: 0097-8930. DOI: 10.1145/325165.325242.

[30] Huynh, D. Q. Metrics for 3D Rotations: Comparison and Analysis. *J. Math. Imaging Vis.* 35.2 (Oct. 2009), 155–164. ISSN: 0924-9907. DOI: 10.1007/s10851-009-0161-2.

[31] Adobe Systems Incorporated. *Mixamo*. 2021. URL: https://www.mixamo.com/ (visited on 03/17/2021).

[32] Magnenat-Thalmann, N., Laperrire, R. and Thalmann, D. Joint-dependent local deformations for hand animation and object grasping. *In Proceedings on Graphics interface'88*. Citeseer. 1988. DOI: 10.5555/102313.102317.

[33] Nintendo Co., Ltd. *The Legend Of Zelda Ocarina Of Time Instruction Booklet*. 1998. URL: https://cdn02.nintendo-europe.com/media/downloads/games_8/emanuals/nintendo_8/Manual_Nintendo64_TheLegendOfZeldaOcarinaOfTime_EN.pdf (visited on 03/07/2021).

[34] Pixar Animation Studios. *Toy Story*. 2021. URL: https://www.pixar.com/feature-films/toy-story (visited on 03/07/2021).

[35] Lewis, J. P., Cordner, M. and Fong, N. Pose Space Deformation: A Unified Approach to Shape Interpolation and Skeleton-Driven Deformation. *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. USA: ACM Press/Addison-Wesley Publishing Co., 2000, 165–172. ISBN: 1581132085. DOI: 10.1145/344779.344862.

[36] Tarini, M., Panozzo, D. and Sorkine-Hornung, O. Accurate and efficient lighting for skinned models. *Computer Graphics Forum*. Vol. 33. 2. Wiley Online Library. 2014, 421–428. DOI: 10.1111/cgf.12330.

[37] Kavan, L., Collins, S., O'Sullivan, C. and Zara, J. Dual quaternions for rigid transformation blending. *Technical report, Trinity College Dublin* (2006).

[38] Alexa, M., Behr, J. and Müller, W. The morph node. *Proceedings of the fifth symposium on Virtual reality modeling language (Web3D-VRML)*. 2000, 29–34. DOI: 10.1145/330160.330172.

[39] Sutherland, I. E., Sproull, R. F. and Schumacker, R. A. A Characterization of Ten Hidden-Surface Algorithms. *ACM Comput. Surv.* 6.1 (Mar. 1974), 1–55. ISSN: 0360-0300. DOI: 10.1145/356625.356626.

[40] Yang, L., Liu, S. and Salvi, M. A survey of temporal antialiasing techniques. *Computer Graphics Forum*. Vol. 39. 2. Wiley Online Library. 2020, 607–621. DOI: 10.1111/cgf.14018.

[41] Yang, L., Nehab, D., Sander, P., Sitthi-amorn, P., Lawrence, J. and Hoppe, H. Amortized supersampling. eng. *ACM transactions on graphics* 28.5 (2009), 1–12. ISSN: 0730-0301. DOI: 10.1145/1618452.1618481.

[42] Halton, J. H. Algorithm 247: Radical-Inverse Quasi-Random Point Sequence. *Commun. ACM* 7.12 (Dec. 1964), 701–702. ISSN: 0001-0782. DOI: 10.1145/355588.365104.

[43] Sobol', I. M. On the distribution of points in a cube and the approximate evaluation of integrals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki* 7.4 (1967), 784–802. DOI: 10.1016/0041-5553(67)90144-9.

[44] Scherzer, D., Yang, L. and Mattausch, O. Exploiting temporal coherence in real-time rendering. *ACM SIGGRAPH ASIA 2010 Courses*. 2010, 1–26. DOI: 10.1145/1900520.1900544.

[45] Zimmer, H., Rousselle, F., Jakob, W., Wang, O., Adler, D., Jarosz, W., Sorkine-Hornung, O. and Sorkine-Hornung, A. Path-Space Motion Estimation and Decomposition for Robust Animation Filtering. *Comput. Graph. Forum* 34.4 (July 2015), 131–142. ISSN: 0167-7055. DOI: 10.5555/2858834.2858848.

[46] Epic Games, Inc. *The Unreal Engine 4 source code*. 2015. URL: https://www.unrealengine.com/en-US/ue4-on-github (visited on 03/09/2021).

[47] Wihlidal, G. *4k Checkerboard in Battlefield 1 and Mass Effect Andromeda*. 2017. URL: https://www.ea.com/frostbite/news/4k-checkerboard-in-battlefield-1-and-mass-effect-andromeda (visited on 03/09/2021).

[48] McCool, M. D. Anisotropic diffusion for monte carlo noise reduction. *ACM Transactions on Graphics (TOG)* 18.2 (1999), 171–194. DOI: 10.1145/318009.318015.

[49] Lottes, T. *TSSAA (Temporal Super-Sampling AA)*. Accessed 2021 via Web Archive: "http://web.archive.org/". URL: http://timothylottes.blogspot.com/2011_04_01_archive.html (visited on 04/06/2021).

[50] Karis, B. High-quality temporal supersampling. *Advances in Real-Time Rendering in Games, SIGGRAPH Courses* 1.10.1145 (2014), 2614028–2615455.

[51] Salvi, M. *An Excursion in Temporal Supersampling*. URL: http://developer.download.nvidia.com/gameworks/events/GDC2016/msalvi_temporal_supersampling.pdf (visited on 04/06/2021).

[52] Andersson, P., Nilsson, J., Salvi, M., Spjut, J. B. and Akenine-Möller, T. Temporally Dense Ray Tracing. *High Performance Graphics (Short Papers)*. 2019, 33–38. DOI: 10.2312/hpg.20191193.

[53] Herzog, R., Eisemann, E., Myszkowski, K. and Seidel, H.-P. Spatio-temporal upsampling on the GPU. *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. 2010, 91–98. DOI: 10.1145/1730804.1730819.

[54] Aalto, T. *Towards Cinematic Quality, Antialiasing in 'Quantum Break'*. 2016. URL: https://www.gdcvault.com/play/1023870/Towards-Cinematic-Quality-Antialiasing-in (visited on 03/18/2021).

[55] Leadbetter, R. *Inside PlayStation 4 Pro: How Sony made the first 4K games console*. URL: https://www.eurogamer.net/articles/digitalfoundry-201

6-inside-playstation-4-pro-how-sony-made-a-4k-games-machine (visited on 03/20/2021).

[56] Keshavarz, B., Hecht, H. and Lawson, B. Visually induced motion sickness: characteristics, causes, and countermeasures. *Handbook of virtual environments: Design, implementation, and applications* (2014), 648–697. DOI: 10.1201/b17360-32.

[57] Rebenitsch, L. and Owen, C. Review on cybersickness in applications and visual displays. *Virtual Reality* 20.2 (2016), 101–125. DOI: 10.1007/s10055-016-0285-9.

[58] So, R. H. and Griffin, M. J. Effects of time delays on head tracking performance and the benefits of lag compensation by image deflection. *Flight Simulation Technologies Conference, New Orleans, Louisiana*. 1991.

[59] LaViola Jr, J. J. A discussion of cybersickness in virtual environments. *ACM Sigchi Bulletin* 32.1 (2000), 47–56. DOI: 10.1145/333329.333344.

[60] Sherman, C. R. Motion sickness: review of causes and preventive strategies. *Journal of travel medicine* 9.5 (2002), 251–256. DOI: 10.2310/7060.2002.24145.

[61] Antonov, M. *Asynchronous Timewarp Examined*. 2015. URL: https://developer.oculus.com/blog/asynchronous-timewarp-examined/?locale=fi_FI (visited on 03/09/2021).

[62] Abrash, M. Down the VR rabbit hole: Fixing judder. (2013). Accessed 2021 via Web Archive: "http://web.archive.org/". URL: http://blogs.valvesoftware.com/abrash/down-the-vr-rabbit-hole-fixing-judder/.

[63] Liu, E. *DLSS 2.0 - Image reconstruciton for Real-time Rendering with Deep Learning*. URL: http://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s22698-dlss-image-reconstruction-for-real-time-rendering-with-deep-learning.pdf (visited on 03/04/2021).

[64] Looman, T. *Exploring DLSS 2.0 in Unreal Engine 4.26*. 2021. URL: https://www.tomlooman.com/dlss-unrealengine/ (visited on 03/18/2021).

[65] Koskela, M., Immonen, K., Mäkitalo, M., Foi, A., Viitanen, T., Jääskeläinen, P., Kultala, H. and Takala, J. Blockwise multi-order feature regression for real-time path-tracing reconstruction. *ACM Transactions on Graphics (TOG)* 38.5 (2019), 1–14. DOI: 10.1145/3269978.

[66] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A. and Salvi, M. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. *Proceedings of High Performance Graphics*. 2017, 1–12. DOI: 10.1145/3105762.3105770.

[67] Amazon Lumberyard. *Amazon Lumberyard Bistro, Open Research Content Archive (ORCA)*. July 2017. URL: http://developer.nvidia.com/orca/amazon-lumberyard-bistro.

[68] Scherzer, D., Yang, L., Mattausch, O., Nehab, D., Sander, P. V., Wimmer, M. and Eisemann, E. A Survey on Temporal Coherence Methods in Real-Time Rendering. *Eurographics 2011 - State of the Art Reports*. Ed. by N. John and B. Wyvill. The Eurographics Association, 2011. DOI: 10.2312/EG2011/stars/101-126.

[69] Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. Numerical recipes in C++. *The art of scientific computing* 2 (1992), 1002.

[70] Karlsson, B. *Renderdoc*. 2018. URL: https://renderdoc.org/ (visited on 03/18/2021).

[71] Apple Inc. *Frame Capture Debugging Tools*. 2020. URL: https://developer.apple.com/documentation/metal/frame_capture_debugging_tools (visited on 03/22/2021).

[72] NVidia Corporation. *NSight*. 2021. URL: https://developer.nvidia.com/nsight-graphics (visited on 03/18/2021).

[73] Microsoft. *PIX on Windows*. 2021. URL: https://devblogs.microsoft.com/pix/introduction/ (visited on 03/22/2021).

[74] Intel Corporation. *Intel® Graphics Performance Analyzers Framework (Intel® GPA Framework)*. 2020. URL: https://intel.github.io/gpasdk-doc/ (visited on 03/22/2021).

[75] *Apitrace*. 2021. URL: https://github.com/apitrace/apitrace (visited on 03/15/2021).

[76] Sutherland, I. E., Sproull, R. F. and Schumacker, R. A. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys (CSUR)* 6.1 (1974), 1–55. DOI: 10.1145/356625.356626.

[77] Luebke, D. A survey of polygonal simplification algorithms. *UNC Chapel Hill Computer Science Technical Report TR97* 45 (1997). DOI: 10.1109/38.920624.

[78] Unity Technologies. *GameObjectRecorder*. 2021. URL: https://docs.unity3d.com/ScriptReference/Animations.GameObjectRecorder.html (visited on 03/15/2021).

[79] Epic Games. *Using Take Recorder*. 2021. URL: https://docs.unrealengine.com/en-US/AnimatingObjects/Sequencer/Workflow/TakeRecorder/UsingTR/index.html (visited on 03/15/2021).

[80] Wang, D. *MineNav: An Expandable Synthetic Dataset Based on Minecraft for Aircraft Visual Navigation*. 2020. arXiv: 2008.08454 [cs.CV].

[81] Williams, R. S. What's Next? [The end of Moore's law]. *Computing in Science Engineering* 19.2 (2017), 7–13. DOI: 10.1109/MCSE.2017.31.

[82] Standard Performance Evaluation Corporation. *SPECviewperf® 2020 benchmark*. 2020. URL: https://www.spec.org/gwpg/gpc.static/vp2020info.html (visited on 03/06/2021).

[83] UL Benchmarks. *Benchmarks and Performance Tests*. 2021. URL: https://benchmarks.ul.com/ (visited on 03/06/2021).

[84] Unigine. *Fair GPU benchmarks*. 2020. URL: `https://benchmark.unigine.com/` (visited on 03/06/2021).

[85] Maxon Computer GMBH. *Cinebench*. 2021. URL: `https://www.maxon.net/en/cinebench` (visited on 03/06/2021).

[86] Bienia, C. and Li, K. *Benchmarking modern multiprocessors*. Princeton University Princeton, NJ, 2011.

[87] Sakalis, C., Leonardsson, C., Kaxiras, S. and Ros, A. Splash-3: A properly synchronized benchmark suite for contemporary research. *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2016, 101–111. DOI: `10.1109/ISPASS.2016.7482078`.

[88] Antochi, I., Juurlink, B., Vassiliadis, S. and Liuha, P. GraalBench: A 3D Graphics Benchmark Suite for Mobile Phones. *SIGPLAN Not.* 39.7 (June 2004), 1–9. ISSN: 0362-1340. DOI: `10.1145/998300.997165`.

[89] Lext, J., Assarsson, U. and Moller, T. A benchmark for animated ray tracing. *IEEE Computer Graphics and Applications* 21.2 (2001), 22–31.

[90] Haines, E. *Neutral File Format*. 1987. URL: `http://netghost.narod.ru/gff/vendspec/nff/index.htm` (visited on 03/08/2021).

[91] Wald, I., Boulos, S. and Shirley, P. Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. *ACM Trans. Graph.* 26.1 (Jan. 2007), 6–es. ISSN: 0730-0301. DOI: `10.1145/1189762.1206075`.

[92] Studios, W. D. A. *Moana Island Scene (v1.1)*. 2018. URL: `https://www.disneyanimation.com/data-sets/?drawer=/resources/moana-island-scene/` (visited on 02/24/2021).

[93] Burley, B., Adler, D., Chiang, M. J.-Y., Driskill, H., Habel, R., Kelly, P., Kutz, P., Li, Y. K. and Teece, D. The Design and Evolution of Disney's Hyperion Renderer. *ACM Trans. Graph.* 37.3 (July 2018). ISSN: 0730-0301. DOI: `10.1145/3182159`.

[94] Wald, I., Cherniak, B., Usher, W., Brownlee, C., Afra, A., Guenther, J., Amstutz, J., Rowley, T., Pascucci, V., Johnson, C. R. et al. Digesting the Elephant–Experiences with Interactive Production Quality Path Tracing of the Moana Island Scene. *arXiv preprint arXiv:2001.02620* (2020). DOI: `arXiv:2001.02620`.

[95] Roosendaal, T. Sintel. *ACM SIGGRAPH 2011 Computer Animation Festival*. SIGGRAPH '11. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2011, 71. ISBN: 9781450309660. DOI: `10.1145/2019001.2019066`.

[96] Blender Foundation. *Blender Cloud*. 2021. URL: `https://cloud.blender.org/welcome/` (visited on 03/20/2021).

[97] Blender Foundation. *Demo Files*. 2021. URL: `https://www.blender.org/download/demo-files/` (visited on 03/15/2021).

[98] Yoon, S.-e. *Korea Advanced Institute of Science and Technology (KAIST) Model Benchmarks*. 2014. URL: `http://sglab.kaist.ac.kr/models/`.

[99] University of North Carolina, T. G. research group at. *UNC Dynamic Scene Benchmarks*. 2018. URL: http://gamma.cs.unc.edu/DYNAMICB/ (visited on 03/08/2021).

[100] Wald, I. *Short bio*. 2019. URL: http://www.sci.utah.edu/~wald/ (visited on 02/24/2021).

[101] Sanzharov, V., Gorbonosov, A., Frolov, V. and Voloboy, A. Examination of the Nvidia RTX. *Proceedings of the 29th International Conference on Computer Graphics and Vision (GraphiCon 2019)*. Vol. 2485. 2019, 7. DOI: 10.30987/graphicon-2019-2-7-12.

[102] Gribble, C. P., Ize, T., Kensler, A., Wald, I. and Parker, S. G. A Coherent Grid Traversal Approach to Visualizing Particle-Based Simulation Data. *IEEE Transactions on Visualization and Computer Graphics* 13.4 (July 2007), 758–768. ISSN: 1941-0506. DOI: 10.1109/TVCG.2007.1059.

[103] *The Stanford 3D Scanning Repository*. 2014. URL: http://graphics.stanford.edu/data/3Dscanrep/ (visited on 02/24/2021).

[104] Blender Foundation. *Shape Keys*. 2021. URL: https://docs.blender.org/manual/en/latest/animation/shape_keys/index.html (visited on 03/20/2021).

[105] Amazon.com Inc. *Demo Files*. 2021. URL: https://aws.amazon.com/lumberyard/downloads/ (visited on 03/15/2021).

[106] Benty, N., Yao, K.-H., Clarberg, P., Chen, L., Kallweit, S., Foley, T., Oakes, M., Lavelle, C. and Wyman, C. *The Falcor Rendering Framework*. Aug. 2020. URL: https://github.com/NVIDIAGameWorks/Falcor.

[107] Winkelmann, M. *Zero-Day, Open Research Content Archive (ORCA)*. Nov. 2019. URL: https://developer.nvidia.com/orca/beeple-zero-day.

[108] Autodesk, I. *Adaptable file format for 3D animation software*. 2021. URL: https://www.autodesk.com/products/fbx/overview (visited on 03/13/2021).

[109] *The GL Transmission Format (glTF) version 2.0*. Version git checkout 23d81c1. 2020. URL: https://github.com/KhronosGroup/glTF/tree/master/specification/2.0 (visited on 03/20/2021).

[110] Autodesk, Inc. *Welcome to the FBX SDK*. 2021. URL: https://help.autodesk.com/view/FBX/2020/ENU/ (visited on 03/13/2021).

[111] MacPherson, G. *Adaptable file format for 3D animation software*. 2020. URL: https://godotengine.org/article/fbx-importer-rewritten-for-godot-3-2-4 (visited on 03/13/2021).

[112] Khronos Group. *COLLADA – Digital Asset Schema Release 1.5.0*. 2008. URL: https://www.khronos.org/files/collada_spec_1_5.pdf (visited on 03/18/2021).

[113] Pixar. *Welcome to the FBX SDK*. 2020. URL: https://graphics.pixar.com/usd/docs/index.html (visited on 03/13/2021).

[114] Nimier-David, M., Vicini, D., Zeltner, T. and Jakob, W. Mitsuba 2: A retargetable forward and inverse renderer. *ACM Transactions on Graphics (TOG)* 38.6 (2019), 1–17. DOI: `10.1145/3355089.3356498`.

[115] Khronos Group. *Ray Tracing In Vulkan*. 2020. URL: `https://www.khronos.org/blog/ray-tracing-in-vulkan` (visited on 03/23/2021).

[116] Wyman, C. and Marrs, A. Introduction to DirectX raytracing. *Ray Tracing Gems*. Springer, 2019, 21–47.

[117] The Khronos Group Inc. *KHR*$_{l}ights_{p}unctual$. 2020. URL: `https://github.com/KhronosGroup/glTF/blob/master/extensions/2.0/Khronos/KHR_lights_punctual` (visited on 03/23/2021).

[118] The Khronos Group Inc. *EXT*$_{l}ights_{i}mage_{b}ased$. 2018. URL: `https://github.com/KhronosGroup/glTF/tree/master/extensions/2.0/Vendor/EXT_lights_image_based` (visited on 03/23/2021).

[119] Zaal, G. *Kloofendal 48d Partly Cloudy*. 2019. URL: `https://hdrihaven.com/hdri/?h=kloofendal_48d_partly_cloudy` (visited on 03/23/2021).

[120] Creative Commons. *About CC Licenses*. 2021. URL: `https://creativecommons.org/about/cclicenses/` (visited on 02/26/2021).

[121] Duske, K. *Cube 2: Sauerbraten*. 2021. URL: `http://sauerbraten.org/` (visited on 02/26/2021).

[122] Clémençon, J.-M. and Konstin. *SuperTuxKart*. 2020. URL: `https://supertuxkart.net/Main_Page` (visited on 02/26/2021).

[123] Wild Fire Games. *0 A.D.* 2021. URL: `https://play0ad.com/` (visited on 02/26/2021).

[124] The Mine Test Team. *MineTest*. 2021. URL: `https://www.minetest.net/` (visited on 02/26/2021).

[125] The Dark Mod Team. *The Dark Mod*. 2021. URL: `https://www.thedarkmod.com/main/` (visited on 02/26/2021).

[126] *dot3 Labs*. 2021. URL: `http://strlen.com/dot3labs/` (visited on 02/26/2021).

[127] Oortmerssen, W. van. *Sauerbraten initial development documentation*. URL: `http://sauerbraten.org/docs/dev/readme_developer.txt` (visited on 02/26/2021).

[128] Id Software. *Quake Github repository*. 2012. URL: `https://github.com/id-Software/Quake` (visited on 02/26/2021).

[129] Id Software. *Doom Github repository*. 2012. URL: `https://github.com/id-Software/Doom` (visited on 02/26/2021).

[130] David, H. *MD5Mesh and MD5Anim files formats*. 2005. URL: `http://tfc.duke.free.fr/coding/md5-specs-en.html` (visited on 02/26/2021).

[131] Duske, Kristian. *Cube 2: Sauerbraten - Model Reference*. URL: `http://sauerbraten.org/docs/models.html` (visited on 02/26/2021).

[132] Quadropolis. *Eternal Valley*. 2013. URL: http://quadropolis.us/node/3747 (visited on 03/01/2021).

[133] Schaufler, G., Dorsey, J., Decoret, X. and Sillion, F. X. Conservative Volumetric Visibility with Occluder Fusion. *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. USA: ACM Press/Addison-Wesley Publishing Co., 2000, 229–238. ISBN: 1581132085. DOI: 10.1145/344779.344886.

[134] Bourke, P. *Wavefront .obj file format specification*. 2011. URL: http://paulbourke.net/dataformats/obj/ (visited on 02/24/2021).

[135] Unity Technologies. *Unity Real-Time Development Platform*. 2021. URL: https://unity.com/ (visited on 03/23/2021).

[136] Facebook Technologies, LLC. *Develop for the Quest Platform*. 2021. URL: https://developer.oculus.com/quest (visited on 03/23/2021).

[137] Unity Technologies. *Unity case studies*. 2021. URL: https://unity.com/case-study (visited on 03/23/2021).

[138] The Khronos Group Inc. *unityGltf*. 2019. URL: https://github.com/KhronosGroup/UnityGLTF (visited on 03/23/2021).

[139] Blender Foundation. *Cycles Open Source Production Rendering*. 2018. URL: https://www.cycles-renderer.org/about/ (visited on 03/09/2021).

A APPENDIX

A.1 Pseudocode of linear blend skinning of vertices in Vulkan vertex shader

```
#define MAX_JOINTS 64

layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec3 inNormal;
layout (location = 2) in vec4 jointIndices;
layout (location = 3) in vec4 jointWeights;

layout (set = 0, binding = 0) uniform UniformBufferObject {
        mat4 projection;
        mat4 model;
        mat4 view;
        vec3 cameraPosition;
} ubo;

layout (set = 1, binding = 0) uniform UBOModel {
        mat4 matrix;
        mat4 jointMatrices[MAX_JOINTS];
} armature;

layout (location = 0) out vec3 outWpos;
layout (location = 1) out vec3 outNormal;

out gl_PerVertex {
        vec4 gl_Position;
};

void main()
{
        mat4 skinMat =
            jointWeights.x * armature.jointMatrices[int(jointIndices.x)] +
            jointWeights.y * armature.jointMatrices[int(jointIndices.y)] +
            jointWeights.z * armature.jointMatrices[int(jointIndices.z)] +
            jointWeights.w * armature.jointMatrices[int(jointIndices.w)];

        vec4 position =
            ubo.model * armature.matrix * skinMat * vec4(inPosition, 1.0);

        outNormal =
            normalize(transpose(inverse(
            mat3(ubo.model * armature.matrix * skinMat)))
            * inNormal);

        outWpos = position.xyz / position.w;
        gl_Position =  ubo.projection * ubo.view * vec4(outWpos, 1.0);
}
```

A.2 Datasets' camera rotation animation in Euler angles

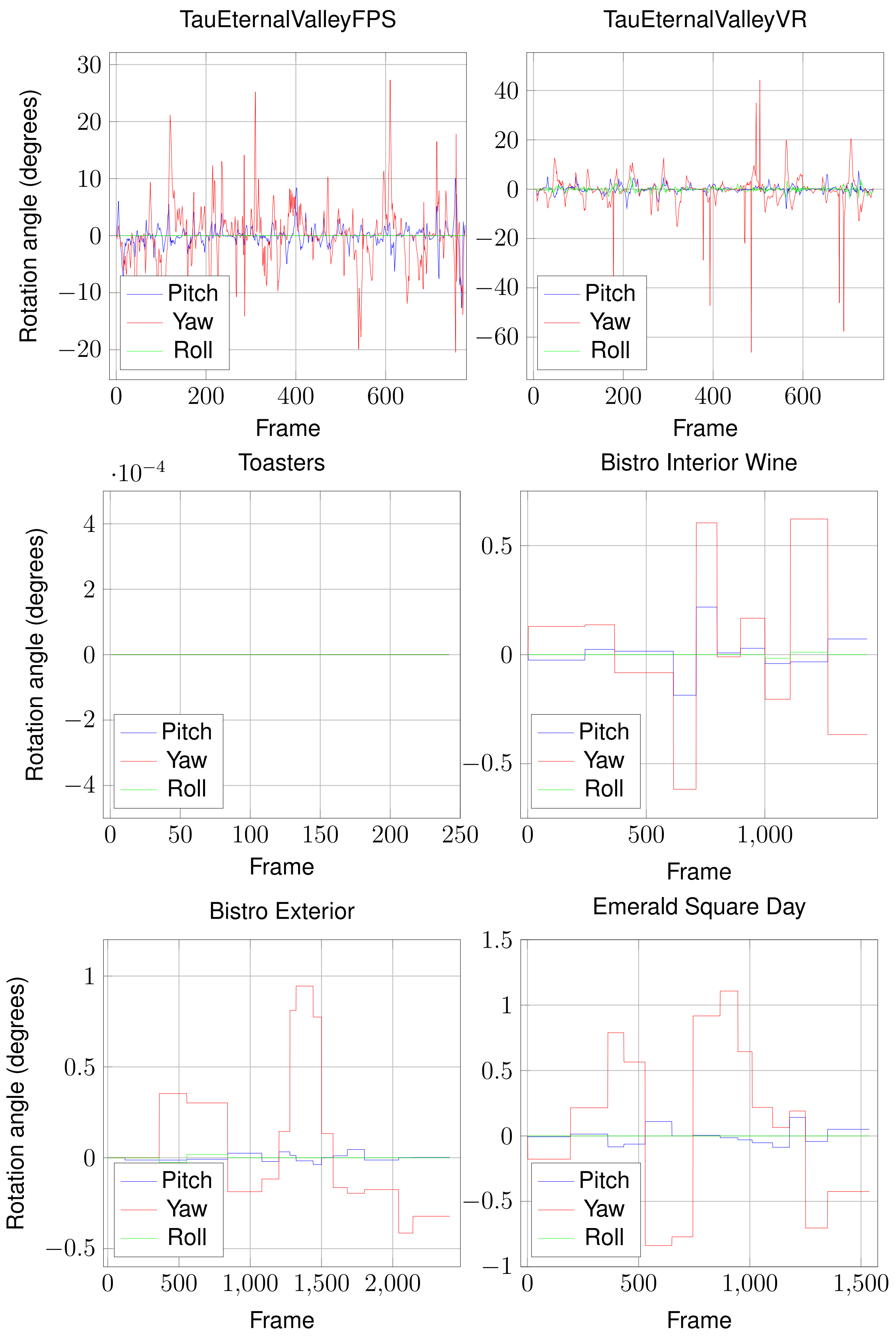

Figure A.1. *Figure of how much the camera rotated per frame in each of its axis*

A.3 Datasets’ camera rotation animation distance

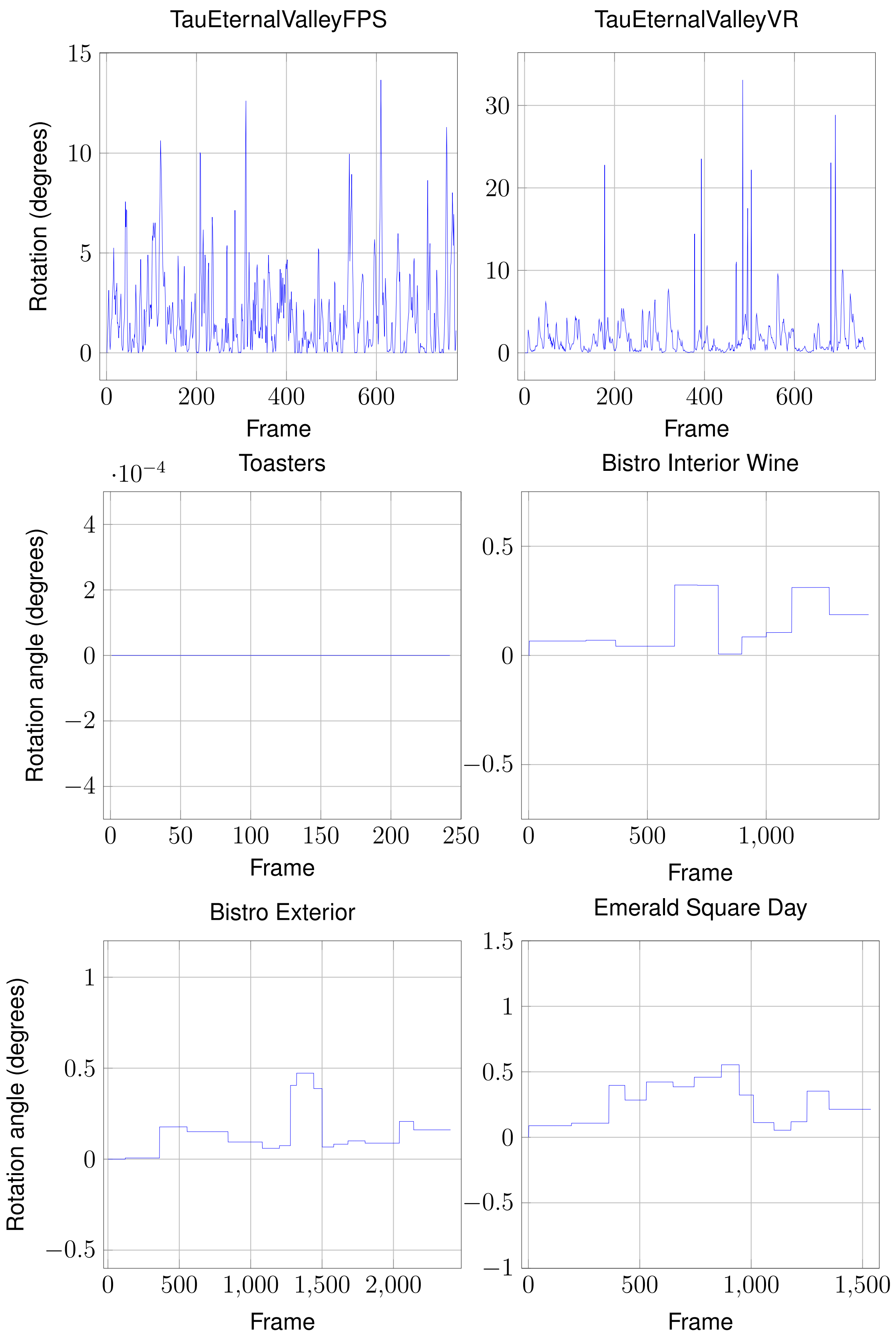

Figure A.2. *Figure of the distance camera rotates in degrees per frame*