International Journal
Information Technology Applications (ITA)

Volume 15, Number 1, July 2026

REAL-TIME DEBUG VISUALIZATION VIA INTER-PROCESS COMMUNICATION IN UNITY EDITOR

Dušan Paulovič

Abstract:

Debugging iterative geometric algorithms in a game engine editor presents a fundamental challenge: when execution is suspended at a breakpoint, the editor becomes non-interactive, making visual inspection of intermediate computational states impossible. This paper presents a debug visualization system that decouples rendering from the main application process through Inter-Process Communication (IPC) via memory-mapped files. Drawing commands - lines, points, meshes, text, and geometric primitives - are serialized using a binary command pattern and transmitted to a secondary Unity Editor instance, which renders them independently while the primary process is suspended. The system exposes a renderer abstraction through the IDrawContext interface, enabling custom rendering backends with no changes to the core system. Zero runtime overhead in production builds is guaranteed by C# conditional compilation. The system was developed and validated in practice during the implementation of geometric collision detection algorithms - including GJK, EPA, SAT, MPR, and incremental convex hull - within a bachelor's thesis project.

Keywords:

Debug visualization, Unity Editor, inter-process communication, memory-mapped file, command pattern, geometric algorithms.

Introduction

Visual debugging is an essential practice in the development of algorithms operating in three-dimensional space. Geometric algorithms such as the Gilbert-Johnson-Keerthi (GJK) distance algorithm, the Expanding Polytope Algorithm (EPA), the Separating Axis Theorem (SAT), and the Minkowski Portal Refinement (MPR) method are inherently spatial: their correctness depends on the relative positions and orientations of geometric structures that are difficult to reason about from numeric output alone. Authoritative treatments of GJK and EPA [1, 2] consistently accompany their algorithmic descriptions with geometric figures, because the correctness of each iteration step can only be assessed visually.

The Unity Editor provides built-in debug drawing utilities, most notably the Gizmos and Handles APIs [3, 4]. These are sufficient for inspecting algorithm state between frames in normal play mode. However, when execution is suspended at a breakpoint, the primary editor instance becomes non-interactive for the purposes of scene inspection [5, 6]. As a result, the developer cannot reliably rotate the scene view, observe newly drawn primitives, or interact with the editor while stepping through an algorithm. This limitation makes visual debugging ineffective at exactly the moments it is most needed.

This paper presents a system that resolves this problem. The core idea is to move rendering to a separate, independent process: a secondary Unity Editor instance dedicated exclusively to visualization. The primary application serializes drawing commands to a named memory-mapped

file; the secondary instance reads and renders these commands continuously, remaining fully interactive regardless of the primary process state. The contribution lies not in a new visualization paradigm, but in a practical integration of out-of-process rendering into the Unity Editor debugging workflow.

The system was designed as a practical development tool and validated through active use during the implementation of several geometric collision detection algorithms for a bachelor's thesis on rigid body physics.

1 Related Work

Several tools and approaches exist for visual debugging within the Unity Editor environment, each with limitations relevant to the problem addressed here.

Unity's built-in Gizmos API allows drawing geometric primitives in the Scene View during edit and play modes [3]. The Handles API, part of the UnityEditor namespace, provides enhanced drawing with more stylistic control [4]. Both APIs are editor-only: they cannot be referenced by runtime code compiled into players. More importantly, both operate within the same process as the editor and are therefore subject to the same thread-freezing limitation during breakpoint debugging: they cannot update while execution is paused.

Logging-based approaches record numeric state to the console or to a file. A developer could in principle build a custom visualizer that reads such logs and renders the geometry, which is conceptually similar to the approach presented in this paper. However, this system does so automatically, using a binary format optimized for spatial data, and renders directly into a secondary Unity Editor instance that is already available to every Unity developer on all supported platforms, requiring no additional tooling.

To the author's knowledge, no widely adopted tool within the Unity ecosystem provides live debug drawing via out-of-process rendering that remains interactive while the primary process is suspended at a breakpoint.

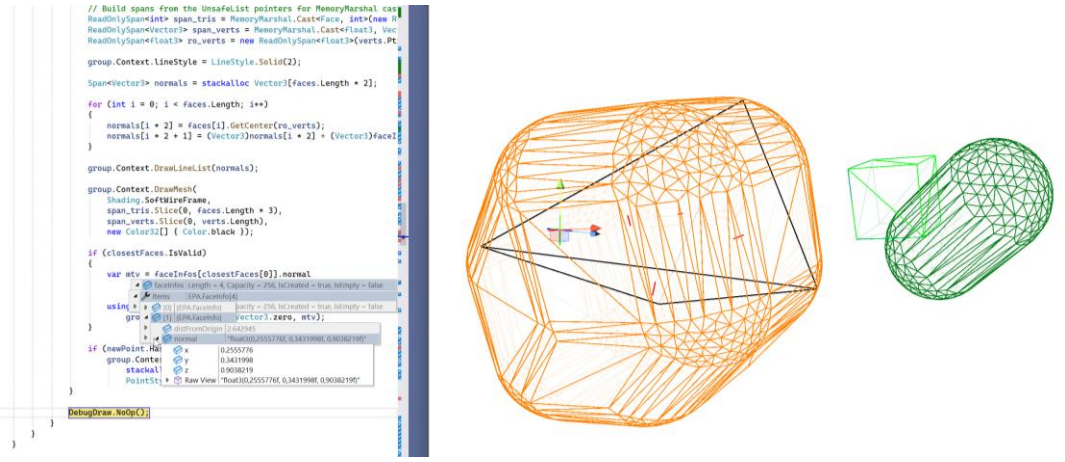

Fig.1. The left side of the image shows Visual Studio paused at a breakpoint, while the right side displays an interactive visualization of the Minkowski difference between a box and a capsule, with the initial simplex of the EPA algorithm contained within it.

2 System Architecture

The system consists of two components: the Producer and the Consumer, communicating through a shared memory region as illustrated in (Fig.2).

The Producer component runs in the developer's primary Unity Editor instance. It consists of the static DebugDraw API class, which accepts drawing calls from application code; the CommandSerializer, which translates drawing calls into binary command data; and the FileCommandHandler with its MMFHandler, which writes serialized commands to a named memory-mapped file.

The Consumer component runs in a secondary Unity Editor instance hosting a dedicated visualization project, identified by the DBG_ prefix in its project name. It consists of the FileCommandDrawer, which periodically reads from the shared memory; the CommandAggregator, which organizes commands by index and group; and the UnityEditorDrawingContext, which executes rendering via Unity's GL API in the Scene View.

Data flows in one direction: primary application → CommandSerializer → memory-mapped file → FileCommandDrawer → CommandAggregator → UnityEditorDrawingContext → Scene View. The secondary instance remains fully interactive, allowing scene rotation and inspection, regardless of the primary process state.

The secondary instance is launched via the Visual Debug / Open Instance editor menu item, which copies the necessary assets into a temporary directory and spawns a new Unity process pointing to that directory. The secondary project automatically detects the DBG_ prefix at startup and initializes the rendering server component (DebugDrawFileServer). Once started, the consumer runs as a long-lived visualization process that persists throughout the debugging session, independent of the execution state of the primary editor. It is started once and reused across recompilations, domain reloads, and debugger attach and detach cycles, amortizing initialization cost across the entire session.

The Unity Editor is reused as a visualization runtime rather than reimplemented, trading higher resource usage for zero additional tooling requirements and immediate integration into the existing development environment. Every Unity developer already has the Unity Editor installed, and it provides scene navigation, camera controls, GL rendering, and cross-platform support without any additional dependencies.

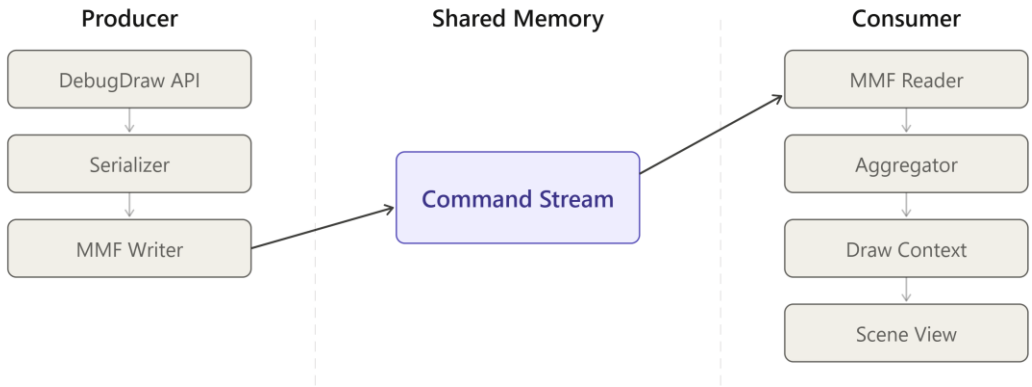

Fig.2. System architecture: Producer serializes commands to shared memory. Consumer reads and renders them independently.

3 Technical Design

3.1 Command Abstraction and Binary Serialization

All drawing operations are represented as commands. The ICommand interface extends ICustomBytes, a serialization contract requiring three members: the ByteSize property, and Serialize and Deserialize methods operating on byte spans. This design cleanly separates command identity from its wire format.

Serialization is performed through a fluent chain of MemoryIterator method calls. MemoryIterator maintains an integer offset into a Span<byte> buffer and provides typed Write and Read methods covering all primitive types as well as specialized handlers for Vector3, Quaternion, Matrix4x4, and UTF-8 strings. This approach is allocation-free, avoids reflection, and produces compact binary representations with predictable sizes.

For example, DrawLineCmd is a 32-byte unmanaged struct containing a CmdHeader (8 bytes: command index and an Operation enum value) and two Vector3 values (12 bytes each). Its serialization chain is (Fig.3):

```
public int ByteSize =>
    MemoryUtil.MoveBySizeOf<CmdHeader>().
    MoveBySizeOfVector3(2);

public int Deserialize(ReadOnlySpan<byte> source) =>
    MemoryUtil.
    Read(source, ref h).
    ReadVector3(source, ref s).
    ReadVector3(source, ref e);

public int Serialize(Span<byte> dest) =>
    MemoryUtil.
    Write(dest, h).
    WriteVector3(dest, s).
    WriteVector3(dest, e);
```

Fig.3. DrawLineCmd serialization.
MemoryUtil operates on MemoryIterator,
which is implicitly convertible to int.

Commands with variable-length payload, DrawLineStripCmd (variable point count) and DrawTextCmd (UTF-8 string), compute ByteSize dynamically using the same MemoryIterator chain in a sizing pass before serialization. This maintains a single source of truth for the byte layout.

Binary serialization was preferred over text-based formats [7] for three reasons: lower serialization latency, smaller memory footprint per command, and deterministic byte layout enabling safe use of unmanaged struct operations and stackalloc buffers. As the producer and consumer are always executed from the same build within a single development environment, version compatibility between command layouts is not enforced: structural changes propagate simultaneously to both sides.

3.2 Renderer Abstraction via IDrawContext

The IDrawContext interface defines the complete drawing vocabulary of the system: DrawLine, DrawLineStrip, DrawLineList, DrawCircleFraction, DrawSphere, DrawWireSphere, DrawCube, DrawWireCube, DrawPointList, DrawText, DrawText3D, and DrawMesh. It also exposes state properties (color, matrix, lineStyle) and scope-based state management methods (SetColorScope, SetMatrixScope, SetLineStyleScope, SetAppearanceScope) that restore the previous state when disposed.

Two IDrawContext implementations are directly involved in the presented Unity Editor pipeline. CommandSerializer translates each method call into a serialized command written to the IPC channel. UnityEditorDrawingContext executes GL calls directly in the Unity Scene View of the consumer process.

```
public interface IDrawContext
{
    IDisposable scope { get; set; }
    Color32 color { get; set; }
    Matrix4x4 matrix { get; set; }
    LineStyle lineStyle { get; set; }

    IDisposable SetFrameScope();
    IDisposable SetMatrixScope(in Matrix4x4 matrix);
    IDisposable SetColorScope(Color32 color);
    IDisposable SetLineStyleScope(LineStyle lineStyle);
    IDisposable SetAppearanceScope(Color32 color, LineStyle lineStyle);

    void SetAppearance(Color32 color, LineStyle lineStyle);
    void DrawLine(in Vector3 p1, in Vector3 p2);
    void DrawCircleFraction(in Vector3 center, float radius,
       in Quaternion rotation, float fraction);
    void DrawLineStrip(ReadOnlySpan<Vector3> points, bool looped);
    void DrawLineList(ReadOnlySpan<Vector3> points);
    void DrawSphere(in Vector3 center, float radius);
    void DrawWireSphere(in Vector3 center, float radius);
    void DrawCube(in Vector3 center, in Vector3 halfExtents, in Quaternion rotation);
    void DrawWireCube(in Vector3 center, in Vector3 halfExtents, in Quaternion rotation);
    void DrawPointList(ReadOnlySpan<Vector3> points, PointStyle pointStyle);
    void DrawText(in Vector3 position, string text, TextStyle style);
    void DrawText3D(in Vector3 position, in Quaternion rotation, string text, TextStyle style);
    void DrawMesh(Shading shading, ReadOnlySpan<int> triangles,
        ReadOnlySpan<Vector3> vertices, ReadOnlySpan<Color32> colors);
}

public interface IAppContext
{
    void Clear();
    void BeginGroup(string name);
    void ClearGroup(string name);
}

public interface IBatchContext : IDrawContext, IAppContext
{
    IDisposable BatchScope();
}
```

Fig.4. Interfaces IDrawContext, IAppContext and IBatchContext.

This separation means that the same user code works identically whether it calls into CommandSerializer (in the primary process) or directly into UnityEditorDrawingContext (in a hypothetical single-process configuration). It also allows developers to implement custom rendering backends by providing an IDrawContext implementation and connecting it to a FileCommandDrawer or CommandAggregator. Examples of possible custom backends include: an SVG exporter for documentation, a recording context for playback debugging (already implemented for unit testing purposes), or a runtime renderer for non-editor builds.

The IDrawContext interface is further extended by IAppContext, which provides Clear, BeginGroup, and ClearGroup operations. IBatchContext combines both, enabling the CommandSerializer to serve as the single entry point for all operations from user code (Fig.4).

3.3 Inter-Process Communication via Memory-Mapped Files

The IPC channel is implemented as a named memory-mapped file (MMF) managed by MMFHandler. The MMF has a fixed capacity of 4 MB, configurable via a single constant. Its layout consists of a 4-byte integer length prefix followed by a UTF-8 encoded sequence of newline-separated hex strings, each representing one serialized command. Although commands are serialized into compact binary form internally, they can be represented as human-readable strings in multiple formats for debugging purposes. Pure hexadecimal is used during debug sessions for full inspectability, while a mixed format (hexadecimal header followed by Base64-encoded payload) is used in production to balance readability with compactness (Fig.5). This representation was chosen deliberately to make the debug stream human-inspectable, easily persistable, and simple to recover from disk. Serialization and transport are not the primary performance bottlenecks in the intended debugging workflow. The current transport format is therefore sufficient for interactive workloads, where the system remains usable even under heavy command loads, and is not intended for high-frequency or high-volume data streaming.

Hexadecimal (debugging purposes):
```
7B00000009000000CDCC8C3FCDCC0CC033335340CDCC8CC00000B0403333D3C0
```

Base64 (not used, but available as an option):
```
ewAAAAkAAADNzIw/zcwMwDMzU0DNzIzAAACwQDMz08A=
```

Header hexadecimal, rest Base64 (production ready):
```
7B00000009000000zcyMP83MDMAzM1NAzcyMwAAAsEAzM9PA
```

Fig.5. Same DrawLineCmd serialized in alternative formats for readability comparison.

Named memory-mapped files were selected over named pipes and TCP sockets for several reasons [8, 9]. MMFs support multiple concurrent readers without protocol overhead or connection management. Their content persists across writer restarts, meaning the visualization remains visible if the primary application reconnects after a crash or reload. Random-access reading is efficient, requiring no sequential scanning.

Both read and write access are synchronized using a NamedMutex. The mutex is acquired for the duration of each read and write operation, ensuring that the consumer always receives a consistent snapshot of the command stream. Because the consumer reads in the editor update loop and write operations are brief, contention is negligible in practice.

On disposal, MMFHandler persists the last written content to disk as a plain text file. This allows the visualization state to survive both primary and secondary editor restarts.

3.4 Command Grouping and Incremental Updates

The CommandAggregator on the consumer side organizes received commands into named groups via BeginGroup and ClearGroup operations, and maintains a SortedDictionary<int, DrawCmdInfo> indexed by command index. This allows selective invalidation of visualization state: when an algorithm's state changes, only the commands belonging to a specific group need to be cleared and replaced.

The ClearGroup operation removes only the commands associated with a given group name from the sorted dictionary, leaving all other visualization intact. This is useful in situations where a temporary visual state needs to be drawn on top of an existing scene and remain visible only within a specific section of code. A recurring pattern across multiple algorithms is creating a local group around each iteration step, so that only the current intermediate state is replaced while persistent annotations remain visible. For example, in GJK the current simplex replaces the previous one each iteration, in EPA the evolving polytope is updated step by step, in MPR the current portal is refreshed, and in SAT the currently tested separating axis is highlighted and cleared before the next one is tested. In (Fig.1) orange Minkowski difference hull and origin cross are persistent while black simplex and red normals are temporary.

3.5 Zero-Overhead Production Builds

All public methods in the DebugDraw static class are decorated with the [Conditional("DEBUG_DRAW")] attribute as shown in (Fig.6). When the DEBUG_DRAW compilation symbol is absent, as in release builds, the C# compiler eliminates all call sites at compile time, producing no runtime overhead: no method calls, no allocations, and no conditional branches. The debug visualization system is completely invisible to the runtime in production.

Additionally, helper methods in algorithm implementations may check Debugger.IsAttached before executing visualization code, ensuring that even in debug builds the visualization is skipped unless a debugger is actively connected, avoiding any performance impact during normal profiling sessions.

```
[Conditional("DEBUG_DRAW")]
public static void DrawRay(Vector3 start, Vector3 direction)
{
    if (GetContext(out var ctx))
        ctx.DrawLine(start, start + direction * _rayLength);
}

[Conditional("DEBUG_DRAW")]
public static void DrawCircleFraction(
    Vector3 center,
    float radius,
    Quaternion rotation,
    float fraction)
{
    if (GetContext(out var serializer))
        serializer.DrawCircleFraction(center, radius, rotation, fraction);
}
```

Fig.6. DebugDraw methods decorated with Conditional attribute.

4 Rendering Pipeline in UnityEditorDrawingContext

This section describes the rendering pipeline provided by the editor-specific implementation of the IDrawContext interface. All geometry is submitted through Unity's GL immediate-mode interface as triangle lists. There are no persistent Mesh objects, no MeshRenderer components, and no render texture intermediaries, every frame data is constructed on the CPU and pushed directly to the GPU. The entry point is a SceneView callback that fires once per Scene View repaint. Within that callback, a single Material.SetPass(0) call activates the custom shader, and all subsequent GL.Begin / GL.Vertex / GL.End calls are batched into a single draw submission per geometric mode change.

The choice of immediate mode is deliberate [10]. Because the geometry may change every frame and is driven by external data, maintaining persistent Mesh objects would introduce unnecessary complexity: meshes would need to be reallocated or resized on every update, vertex buffers would need to be tracked per object, and the Unity object lifecycle would add overhead. Immediate mode bypasses all of this at the cost of re-submitting every vertex each frame, which is acceptable given that debug overlays are not performance-critical paths.

4.1 The Shader

The entire renderer relies on a single shader with one pass. The vertex shader performs only the standard clip-space transform via UnityObjectToClipPos, and the fragment shader returns the per-vertex color unchanged, making it a pure conduit for whatever geometry and color data the CPU produces. The relevant render-state declarations are:

```
Blend SrcAlpha OneMinusSrcAlpha
ZWrite [_ZWrite]
ZTest LEqual
ZClip False
Cull Off
```

Alpha blending is essential because thick lines are drawn as flat quads that can overlap each other and scene geometry; without it, overlapping segments would produce hard edges at their boundaries. ZClip False prevents Unity from discarding geometry at the near plane: a line segment whose endpoint lies behind the camera would otherwise be partially clipped, but for debug visualization such segments should remain fully visible as outgoing rays. Cull Off is required because a line quad that is nearly edge-on to the camera has a surface normal that can flip relative to the view direction; backface culling would then discard the triangle entirely. ZWrite is exposed as a shader property so that it can be toggled per draw call.

4.2 Camera Abstraction and Unit Conversions

Rendering operates on a minimal camera abstraction that is populated from the active scene Unity camera each frame. It provides a stable interface for accessing view and projection parameters during rendering.

4.2.1 The CameraInfo Snapshot

Rather than querying the camera object repeatedly during a frame (which involves managed property accesses and potential Unity-side validation overhead) all camera state is captured once into an unmanaged CameraInfo struct at the beginning of each repaint. The struct stores the camera's

localToWorldMatrix, the six frustum planes, and the scalar properties orthographic, orthographicSize, fieldOfView, and pixelHeight. Direction vectors and position are read directly from the columns of the stored matrix using unsafe pointer arithmetic, returning read-only references to the actual memory occupied by each column and avoiding any Vector3 copy construction.

4.2.2 World-Space Size of a Screen Unit

The renderer supports four measurement units: World (raw world-space units), Pixel (screen pixels), DIP (device-independent pixels, 1 DIP = DPI/96 pixels), and Em (typographic units, 1 Em = DPI/72 pixels). To draw a line of thickness 2 DIP or text of size 14 Em at any depth in the scene, the system must know the world-space extent corresponding to one such unit at the specific point being drawn.

For a perspective camera, the derivation follows from the standard pinhole projection model [10]. The visible height of the frustum slice at depth z (measured along the camera forward axis) is $H(z) = 2z \cdot \tan(FoV/2)$. Since this height spans pixelHeight pixels, the world-space size of one pixel is:

$$unitsPerPixel(z) = 2z \cdot \tan(FoV / 2) / pixelHeight$$

In code, z is the projection of the vector from the camera position to the query point onto the camera forward axis. For an orthographic camera the formula simplifies to $unitsPerPixel = 2 \cdot orthographicSize / pixelHeight$, independent of depth. The final value is multiplied by the pixel-equivalent of the requested unit type (DPI/96 for DIP, DPI/72 for Em). A line specified as “2 DIP thick” therefore subtends exactly 2 device-independent pixels on screen regardless of its world-space depth, making overlays visually stable under any camera movement or zoom.

4.3 Thick Line Rendering

A one-pixel line can be drawn with GL.LINES, but the effective line width is not a reliable cross-platform mechanism for producing arbitrary thick lines [11]. Producing a line of arbitrary visual thickness therefore requires explicit geometry. Each line segment is replaced by a camera-facing quad: a flat rectangle whose long axis aligns with the segment direction and whose short axis is perpendicular to both the segment direction and the current view direction. The two segment endpoints define the centerline; at each endpoint a perpendicular offset vector is computed, and the four quad corners are the endpoints each displaced by ±offset:

```
quad[0] = A + offset_A      quad[1] = B + offset_B
quad[3] = A − offset_A      quad[2] = B − offset_B
```

This quad is submitted as two triangles: (0, 1, 2) and (0, 2, 3). The offset magnitude equals halfThickness at each endpoint, which is derived from the line thickness value converted to world space via the unit conversion described in Section 4.2.2.

4.3.1 Computing the Perpendicular Offset

The offset vector must be perpendicular to the segment as it appears on screen and must lie in a plane visible to the camera, so that it actually produces width when rasterized. For a perspective camera the view direction at a world point is the normalized vector from the camera position to that point — not the global camera forward, which would introduce directional error for points away from screen centre. Given the local view direction and the segment direction, the screen-space perpendicular is computed by projecting the segment direction onto the plane perpendicular to the view direction and taking the cross product with the view direction:

```
Vector3 projected = Vector3.ProjectOnPlane(dir, viewForward);
Vector3 offset    = Vector3.Cross(projected, viewForward)
                      .normalized * halfThickness;
```

ProjectOnPlane(v, n) computes $v - (v \cdot \hat{n})\hat{n}$, removing the component of the direction vector along the view axis. The cross product of the projected direction with the view direction is then perpendicular to both — pointing sideways on screen regardless of the 3D orientation of the line.

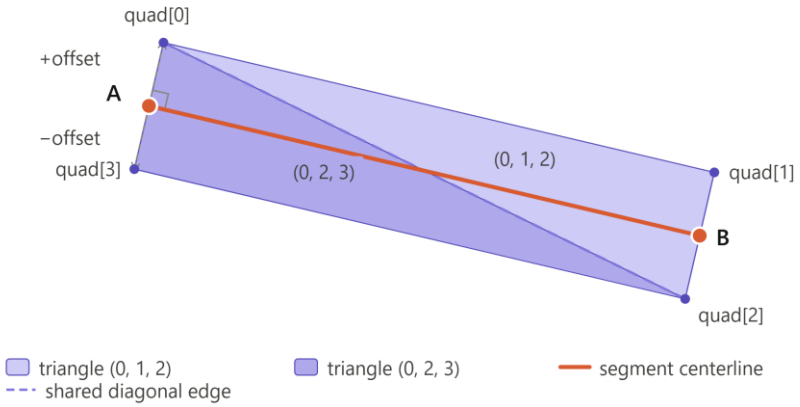

Fig.7. Two triangles per quad, one quad per segment.

4.3.2 Miter Joins

When two segments share a vertex in a polyline, a naive approach produces a gap or overlap between adjacent quads depending on the join angle. A miter join [12] resolves this by extending both quad edges until they intersect at a single point, filling the corner exactly. At a join vertex with incoming projected direction d_A and outgoing direction d_B, the miter direction is the normalised bisector:

```
Vector3 tangent = (dirA + dirB).normalized;
Vector3 miter   = Vector3.Cross(tangent, viewForward).normalized;
```

The miter offset length equals halfThickness / cos θ, where θ is the half-angle between the miter and either segment normal. This follows from the requirement that the miter endpoint lies on both quad edges simultaneously as shown in (Fig.8). For shallow angles $\cos\theta \approx 1$ and the miter approaches halfThickness; for sharp corners $\cos\theta \to 0$ and the length grows without bound. To prevent arbitrarily long spikes, the length is clamped to a configurable multiple of halfThickness; if exceeded, the join falls back to a bevel where both quads terminate at their own segment normals, leaving a small triangular gap that is visually acceptable for corners sharper than roughly 10–15°. A further degenerate case arises when the two projected directions are nearly antiparallel (±180° hairpin): the tangent vector has near-zero magnitude and cannot be normalised, so the outgoing segment normal is used directly.

4.3.3 Segments Facing the Camera

When a segment is nearly parallel to the view direction — pointing roughly toward or away from the camera — it projects to near-zero screen length and the cross-product offset computation

becomes numerically unstable. The condition is detected by testing whether the squared cosine of the angle between the segment direction and view direction is close to 1. When this holds, no quad is emitted; instead, both endpoints are rendered as filled circles with radius equal to halfThickness. This is geometrically correct: a cylindrical line of finite radius viewed end-on appears as a disc of that radius.

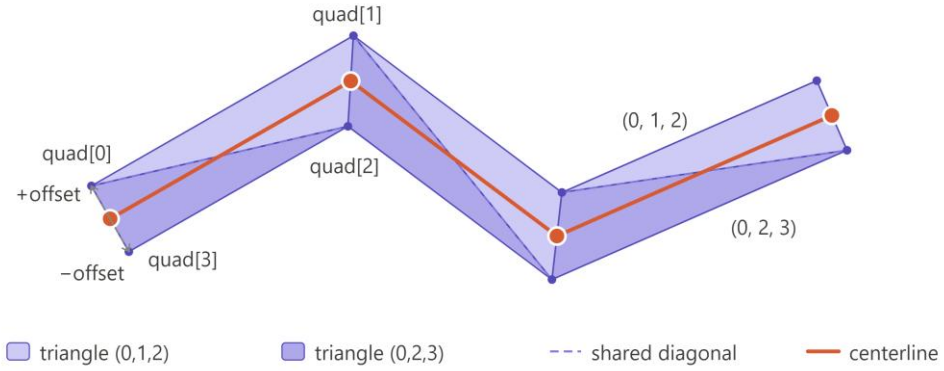

Fig.8. Adjacent quads share miter points at polyline joints.

4.3.4 Round Endpoint Caps

At the two terminal endpoints of a polyline there is no adjacent segment to form a join with. A filled disc is drawn at each endpoint as a triangle fan: its vertices are precomputed as a fixed set of points on the unit circle oriented to face the camera, then scaled by halfThickness and translated to the endpoint position. The same routine is reused for the degenerate end-on segment case (Section 4.3.3) and for rendering Dot pattern elements in dashed lines (Section 4.3.5).

4.3.5 Dashed and Dotted Patterns

Dashed patterns cannot be applied at the segment level of a polyline because the dash/gap rhythm must be consistent in perceived screen distance regardless of how the underlying polyline is sampled [12]. The approach is to first resample the polyline at a uniform interval and then map the resulting point sequence to pattern elements. Resampling operates either in world space (for World-unit spacing) or in screen space (for Pixel and DIP spacing). Screen-space resampling projects each segment to 2D screen coordinates, inserts new sample points at uniform pixel intervals, and recovers the corresponding 3D world position from the interpolated screen position and linearly interpolated depth. This ensures that dashes of a given DIP length maintain constant visual size at any scene depth. The resampled points are consumed by a state machine: a Dot element draws a circle; a Dash element draws a quad spanning two consecutive points; a LongDash element draws two connected quads spanning three points. The pattern repeats cyclically and is encoded in a single byte using two bits per element for up to four elements. For example, the DashDot line pattern is encoded as the value 6 (binary 00 00 01 10), where 10 represents Dash, 01 represents Dot, and 00 terminates the pattern, as shown in (Fig.9).

4.4 Text Rendering

Text is rendered using pre-triangulated vector fonts rather than signed distance field textures or rasterised bitmap atlases. Each glyph is stored as a set of 2D vertices and a triangle index list that

defines the filled area of the glyph outline. Fonts are distributed as GZip-compressed binary files, one per typeface variant, and parsed on first access into an in-memory dictionary keyed by Unicode character. Glyph coordinates are normalised to an emSize of 100 — the cap height of a standard uppercase letter spans 100 units — so the scale factor at render time is the requested font size divided by 100, multiplied by the world-space extent of one screen unit at the text anchor point (Section 4.2.2). Each vertex coordinate axis is stored as an unsigned byte (0–255), scaled by the glyph's own bounding box dimension: coord = raw / 255 × max, where max is the per-glyph width or height stored in the file header, as shown in (Fig.7).

Table 1. Predefined line patterns and their binary encoding.

Name	e[3] e[2] e[1] e[0]	Byte	Preview
Solid	00 00 00 00	0x00	———
Dot	00 00 00 01	0x01	• • • • • • • •
Dash	00 00 00 10	0x02	— — — — — —
DashDot	00 00 01 10	0x06	—• —• —• —•
DashDotDot	00 01 01 10	0x16	—••—••—••—
LongDash	00 00 00 11	0x03	—— —— —— ——
LongDashDot	00 00 01 11	0x07	——•——•——•—
LongDashDotDot	00 01 01 11	0x17	——••——••——••—
LongDashDash	00 00 10 11	0x0B	—— — —— — ——
LongDashDashDotDash	10 01 10 11	0x9B	—— —•— —— —•—

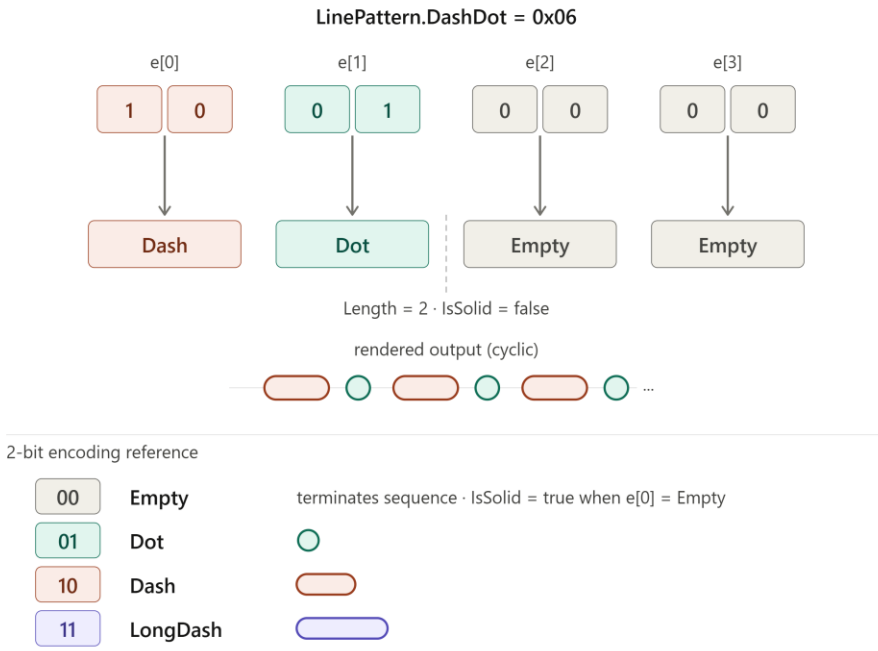

Fig.9. Line patterns are represented by a single byte consisting of up to four two-bit elements.

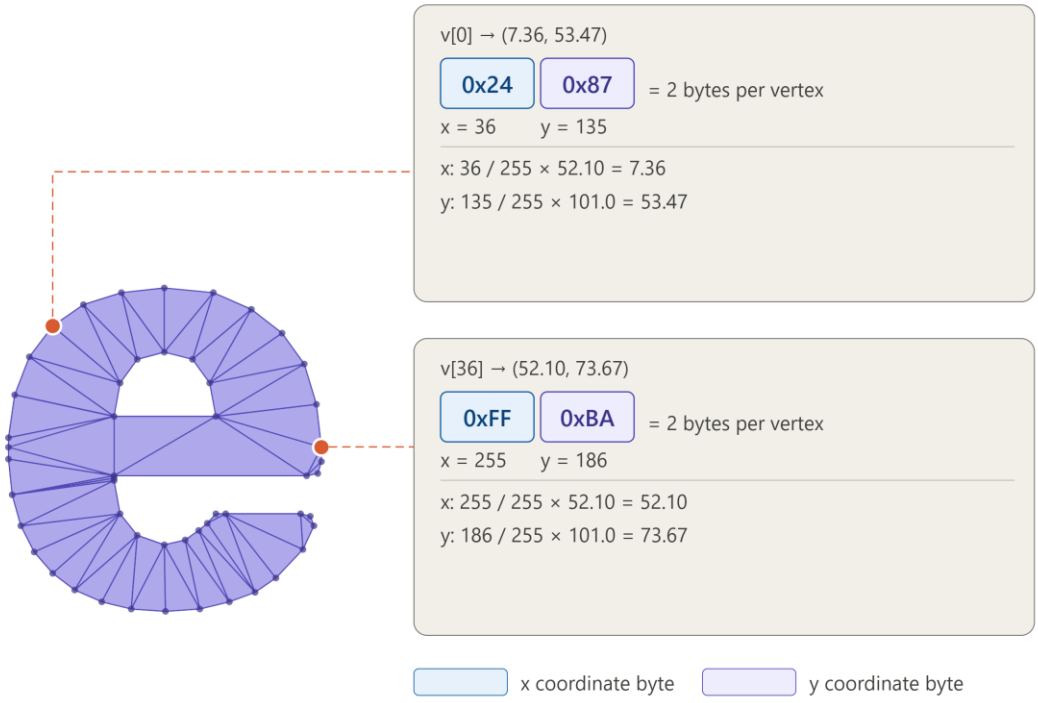

Fig.10. Each vertex coordinate axis is stored as an unsigned byte (0–255).

4.4.1 Layout

Before any vertex data is produced, a measurement pass iterates over the input string and accumulates the total bounding box of the text block. Each glyph contributes its stored width multiplied by the scale factor, plus a configurable letter-spacing value. Space characters advance by the average glyph width; tab characters snap to the next multiple of four average widths; newline characters reset the horizontal position and increment the line counter. From the computed bounds and the style alignment flags, a base offset is derived that positions the text block so the anchor point corresponds to the requested alignment corner or edge. An additional alignment offset scalar pushes the block further along the alignment direction by a fixed screen-space amount, which is useful for preventing a label from overlapping the annotated point.

4.4.2 Glyph Rendering

Each glyph's vertices are transformed from the font's local 2D coordinate system into world space. The y-axis is negated and offset by the glyph height because font coordinate systems place the origin at the top-left with y increasing downward. In billboard mode, the rotation applied to each vertex is taken from the camera transform, ensuring the text plane always faces the viewer. In 3D mode, an explicit rotation quaternion is supplied by the caller. Because the scale depends only on the anchor point, all glyphs within a string are scaled uniformly and text does not deform under perspective. The fill is rendered by iterating over the triangle list and emitting three coloured vertices per triangle directly to the GL stream.

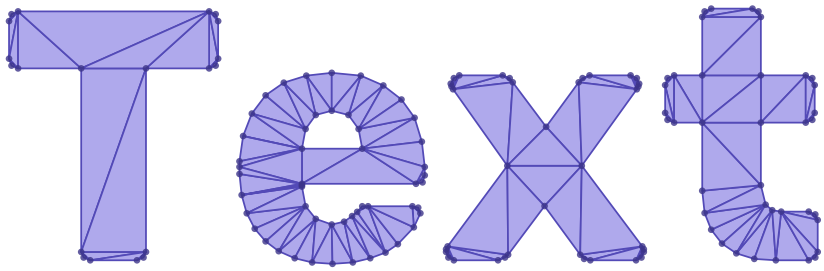

Fig.11. Word "Text" rendered from the binary font format.
Each glyph is decomposed into triangles; vertices are shown as dark dots.

4.4.3 Vector Outline

When an outline width greater than zero is requested, the glyph outline is derived analytically from the triangulation rather than stored separately. The boundary edges of the triangulation — edges belonging to exactly one triangle — collectively form all contour curves of the glyph. They are identified by counting edge occurrences across all triangles: each canonical edge (smaller vertex index first) is counted, and edges with a count of one are boundary edges. These are assembled into closed contours by building an adjacency list and performing an iterative graph walk starting from each unvisited boundary vertex. Each contour — an outer boundary or interior hole — is submitted to GLDrawPolyline as a closed polyline with the outline colour and stroke width, inheriting the full miter join and round cap logic from Section 4.3. Outline quality is resolution-independent since it derives from the same vector data as the fill.

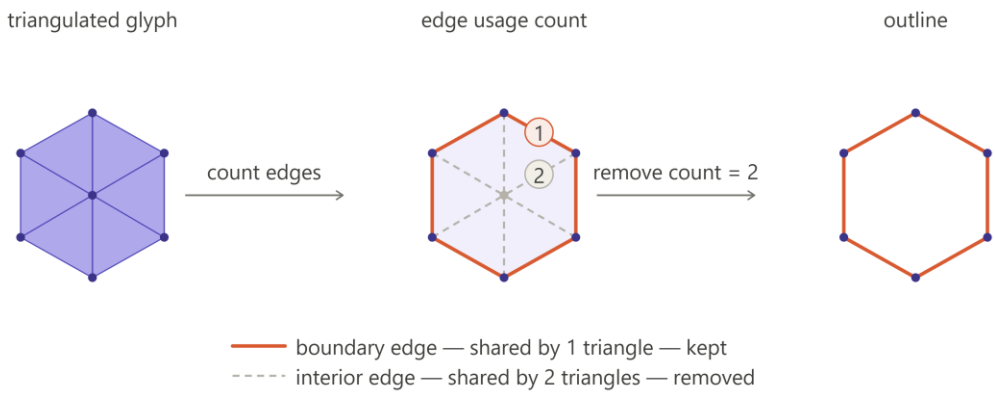

Fig.12. The diagram shows how a glyph outline is extracted from a triangulation.

4.5 Mesh Rendering

Mesh rendering is exposed through a dedicated method for submitting indexed triangle geometry to the renderer:

```
void DrawMesh(
    Shading shading,
    ReadOnlySpan<int> triangles,
    ReadOnlySpan<Vector3> vertices,
    ReadOnlySpan<Color32> colors);
```

This method defines the contract for passing triangle index data together with per-vertex positions and colors. The triangles buffer encodes indexed geometry in triplets, while vertices and colors provide the corresponding attributes. The shading parameter selects the shading mode applied during rasterization. The implementation is responsible for assembling triangles, evaluating their visibility and orientation, and applying the selected shading model in a consistent manner.

4.5.1 Shading Modes

Three shading modes are supported, following the classical taxonomy [10]. Unlit mode renders only front-facing triangles with uniform vertex colours and no lighting computation. A triangle is front-facing when the dot product of its geometric normal with the view direction from the camera to the triangle centroid is positive; the per-triangle view direction is computed individually to account for perspective. Flat shading uses the same dot product as a per-triangle brightness multiplier applied to the vertex colours, producing a faceted appearance useful for inspecting surface orientation. Smooth shading derives per-vertex normals automatically by welding vertices that share the same world-space position within a configurable threshold and accumulating face normals from all adjacent triangles within each welding group. After normalisation, these welded normals produce smooth brightness interpolation across triangulated surfaces without any additional data from the caller.

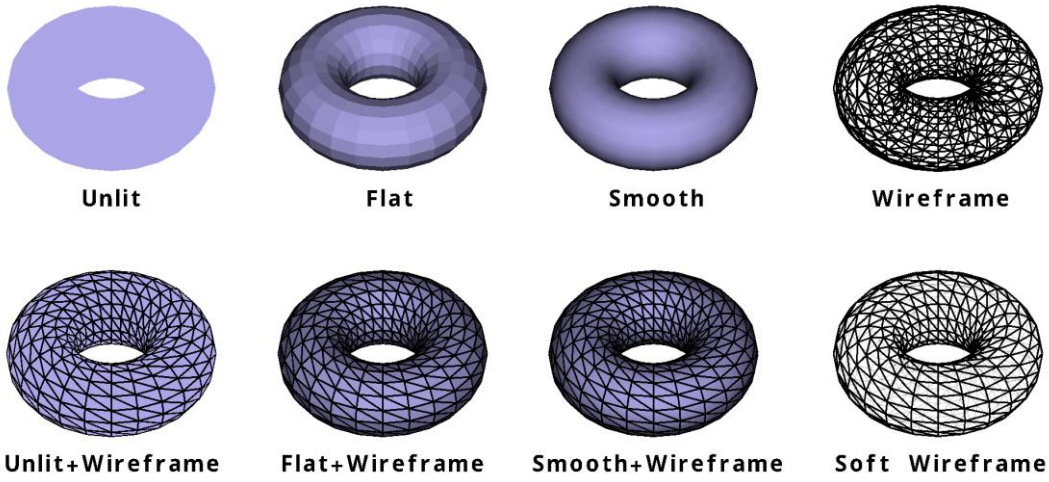

Fig.13. Preview of available mesh smoothing modes.

4.5.2 Wireframe Overlay

When the WireFrame flag is set, mesh edges are rendered as thick lines drawn slightly in front of the surface to avoid z-fighting. The edge extraction step enumerates all triangle edges in canonical form (smaller vertex index first), tags each with a backface flag based on the facing of its parent triangle relative to the camera, and sorts the resulting array. A single linear scan then retains only the first occurrence of each unique edge:

```
Array.Sort(edgesArr.Array, 0, edgeCount);
for (int i = 0; i < edgeCount; i++)
    if (i == 0 || edge.value != edges[i-1].value)
        edges[uniqueCount++] = edge;
```

When the SoftBackfaces flag is enabled, back-facing edges are drawn at 50 % opacity and half the nominal line thickness. This reveals the rear structure of a mesh as a visually subordinate wireframe without obscuring the front, giving an X-ray-like appearance.

5 Limitations and Future Work

The current functionality is limited to simple 3D geometric primitives and text. Rendering features such as 2D graphics in screen coordinates, 3D text with depth, textures, and more complex mesh operations are not yet supported. The IDrawContext abstraction provides a clear extension point for adding these capabilities. The system is designed for developer debugging workflows and is not intended as a general-purpose visualization or rendering framework at this stage.

A planned improvement is the integration of Unity Burst Compiler for performance-critical rendering paths on the consumer side, or alternatively the use of a dedicated external C++ library for these workloads. Burst compilation can significantly reduce the overhead of processing large numbers of draw commands by compiling C# code to highly optimized native code using an LLVM-based optimization pipeline [13]. The implementation is already compatible with the Burst compiler, however, Burst is distributed as a separate package of approximately 500 MB. Since the consumer instance is created as a fresh Unity project, including Burst would substantially increase installation size and setup time. This requires a solution for sharing compiled libraries between the parent project and the consumer instance without redundant installation.

An alternative approach using a native C++ DLL would result in a much smaller footprint. However, the drawback is that the library would need to be compiled separately for each target platform, which increases build and maintenance complexity.

A further planned feature is the ability to share the entire scene state from the primary project to the consumer instance, not just explicit debug draw calls, but the full set of GameObjects, transforms, and meshes present in the primary scene at the time of a breakpoint. This would allow the developer to inspect the complete scene context without any manual instrumentation. This feature requires additional research into what scene data can be efficiently serialized and transmitted via the existing IPC channel, and what Unity APIs are available for reconstructing scene state in the consumer instance.

A standalone native viewer, independent of the Unity Editor, is also in early development. This would allow the consumer to run as a lightweight application rather than a full Unity Editor instance, reducing startup time and resource consumption. The standalone viewer would still be implemented as a Unity player, but its managed code would be compiled ahead-of-time into native code using IL2CPP, eliminating JIT compilation at runtime [14]. As a result, the solution would not require the Unity Editor at runtime. However, the Unity Burst Compiler may still be employed for performance-critical workloads, particularly where data-oriented optimizations such as SIMD vectorization provide measurable benefits.

6 Conclusion

This paper presented a debug visualization system for Unity that decouples rendering from the main application process using inter-process communication via memory-mapped files. The system enables live, interactive visual inspection of algorithmic state during breakpoint debugging,

a capability not provided by existing in-process debug drawing tools, which freeze alongside the application thread. The primary contribution lies in integrating out-of-process rendering into the Unity Editor debugging workflow in a way that requires no additional tooling beyond the Unity Editor itself.

The key technical contributions are: a binary command serialization framework with a clean ICustomBytes contract and allocation-free MemoryIterator chains; a renderer abstraction through the IDrawContext interface enabling custom rendering backends; an IPC channel via named memory-mapped file with mutex-synchronized writes; command grouping with incremental invalidation for efficient state updates; and zero production overhead through [Conditional] attribute-based elimination.

The system was validated in practice as an essential tool during the implementation and adaptation of established geometric collision detection algorithms, including GJK, EPA, SAT, and MPR, as well as an incremental convex hull used as a ground-truth reference, as part of a bachelor's thesis. The collision detection algorithms were implemented independently, with development informed by standard algorithmic descriptions [1,2,10] and, where necessary, by publicly available reference materials and implementations. The incremental convex hull implementation was based on standard algorithmic principles [15] and partially derived from a publicly available reference implementation [16], which influenced its structure and naming. The visual feedback it provided was directly instrumental in identifying and resolving correctness issues that would have been significantly harder to diagnose through numeric logging alone.

Acknowledgement

The author would like to express his sincere gratitude to Ing. Juraj Štefanovič, PhD. for his professional guidance and valuable advice during the development of the bachelor's thesis project in which this system was created. The author also wishes to express his appreciation to his family for their patience, understanding, and support during the completion of this project.

References

[1] Ericson, C. (2005). Real-Time Collision Detection. Morgan Kaufmann. ISBN 978-1-55860-732-3.
[2] van den Bergen, G. (2003). Collision Detection in Interactive 3D Environments. Morgan Kaufmann. ISBN 978-1-55860-801-6.
[3] Unity Technologies (2025). Gizmos and Handles. Unity Manual. Retrieved April 2026, from https://docs.unity3d.com/Manual/gizmos-and-handles.html
[4] Unity Technologies (2025). Handles. Unity Scripting API. Retrieved April 2026, from https://docs.unity3d.com/ScriptReference/Handles.html
[5] Stall, M. (2008). Why you sometimes get a bogus ContextSwitchDeadLock MDA under the debugger. Microsoft Developer Blog. Retrieved April 2026, from https://learn.microsoft.com/en-us/archive/blogs/jmstall/why-you-sometimes-get-a-bogus-contextswitchdeadlock-mda-under-the-debugger
[6] Microsoft Corporation (2024). Debugging Managed Code Using the Windows Debugger. Microsoft Learn. Retrieved April 2026, from https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-managed-code
[7] Google LLC (2024). Protocol Buffers Developer Guide. Retrieved April 2026, from https://protobuf.dev/overview/
[8] Microsoft Corporation (2024). Memory-Mapped Files. .NET Documentation. Retrieved April 2026, from https://learn.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files
[9] Stevens, W. R., Fenner, B., & Rudoff, A. M. (2004). UNIX Network Programming, Vol. 2: Interprocess Communications (2nd ed.). Addison-Wesley. ISBN 978-0-13-081081-6.

[10] Akenine-Möller, T., Haines, E., & Hoffman, N. (2018). Real-Time Rendering (4th ed.). A K Peters/CRC Press. ISBN 978-1-138-62700-0.
[11] Kilgard, M. J., & Bolz, J. (2012). GPU-accelerated Path Rendering. ACM Transactions on Graphics, 31(6). https://doi.org/10.1145/2366145.2366191
[12] W3C (2011). Scalable Vector Graphics (SVG) 1.1 (Second Edition), Section 11.4. Retrieved April 2026, from https://www.w3.org/TR/SVG11/painting.html
[13] Unity Technologies (2025). Burst Compiler Documentation. Retrieved April 2026, from https://docs.unity3d.com/Packages/com.unity.burst@latest
[14] Unity Technologies (2025). Scripting Backends – IL2CPP. Retrieved April 2026, from https://docs.unity3d.com/Manual/scripting-backends-il2cpp.html
[15] Voxagon (2013). Convex Hulls Revisited. Retrieved April 2026, from https://blog.voxagon.se/2013/03/24/convex-hulls-revisited.html
[16] SpexGuy (n.d.). MinkowskiHull3D. GitHub repository. Retrieved April 2026, from https://github.com/SpexGuy/MinkowskiHull3D

Authors

Dušan Paulovič
Pan-European University,
Faculty of Informatics, Bratislava, Slovakia
dusan.paulovic@gmail.com
Software engineer focused on developer tools and complex desktop applications. He works primarily with C++ and C#, specializing in code parsing, IDE integration, and user interface design. He has contributed to widely used development tooling, with experience spanning CAD/GIS systems, 3D visualization, and performance-oriented software design.