Volume xx (20yy), Number z, pp. 1–5

Novel Applications of a “Scrolling”-Algorithm

Abstract

We present and propose novel applications of a scrolling-algorithm which was often used in amiga games. It can be used both in 2D (tile- or pixel-based) and in the future perhaps in 3D (brick- or voxel-based). The current 2D-implementation is pixel-based and therefore it can operate on hand crafted level bitmap images instead of using tiles. In the future it would be interesting to use it for caching/scrolling of voxel data in realtime 3D voxel ray-tracing systems because like on the amiga the video memory is limited. But this limitation could be overcome for some realtime applications where the camera position does not have to jump arbitrarily but only continuous movement of the camera is needed to be done quickly. Camera rotation still could be done arbitrarily then.

1. Introduction

The amiga home-computers of the 1980s and beginning 1990s were well known for realtime games where 2D playfields ([Oos21]) were moved across the CRT screen. As described in the book [Cop25] many programming beginners tried to implement games themselves on these systems beginning with a genre sometimes called “side-scroller” or “shoot em up”. One of the key parts of these games is the so called “scrolling”. In our implementation we implemented such a side-scrolling-game. But we don’t use so called “tiles” or maps of “tiles” like it was usually done. Instead we demonstrate that the video memory of a stock amiga500 computer with 0,5 MB “chip” RAM and 0,5 MB “fast” RAM is sufficient to implement such a game with freely hand crafted background levels. Inspired by this we develop ideas how its scrolling-algorithm could be used today for caching in voxel based 3D realtime raytracing.

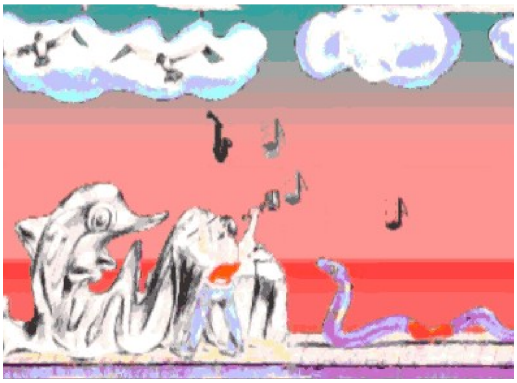

Figure 1: *screenshot of 2D scrolling game implementation*

2. Approach

2.1. Memory Efficient Scrolling Implementation in 2D

The key idea of the scrolling-algorithm relies on the fact that the user of such a game does not see a difference between exact copies of the screen-bitmap while playing the game. In the game the screen seems to be simply scrolled from right to left. The level bitmap is as wide as 5 screens (320*5 pixels in 16 colors). The height is 200 pixels (the height of NTSC without “interlace” on the amiga). The scrolling-algorithm is only needed to save chip memory on the amiga500. If there was no such limited amount of memory the amiga could almost do it in hardware by simply setting some hardware registers. But then the whole large level bitmap would have to be stored 2 times (in 32 colors because of enemy blitting plane) - once as front- and once as back-buffer. This is because amiga games strongly rely on “double buffering” to avoid flickering screens. The blitter has much more time to draw additional game objects (at least 1 video beam period of the whole screen) without flickering results if double buffering is used. With the scrolling-algorithm smaller 32 color front- and back-buffers - which have only twice of the screen width (640*200 pixels) - can be displayed and scrolled with hardware-based register setting on the screen. To still see the whole level throughout the game on the screen, one vertical screen-line per frame is copied with the blitter chip of the amiga at right of the visible part of the buffer and also at left of visible part of the buffer. This does not cost much render-time since the blitter is used and it is only one screen line - 200 pixels in y direction. After 320 pixels of scrolling in x-direction the hardware scrolling registers of the amiga can simply jump back to the first half of the buffer. The player of the game does not recognize this jump, as the two halves of the buffer have been filled with exactly the same pixels! Nevertheless the position from which the screen-lines are copied can move continuously through the larger level bitmap throughout the game.

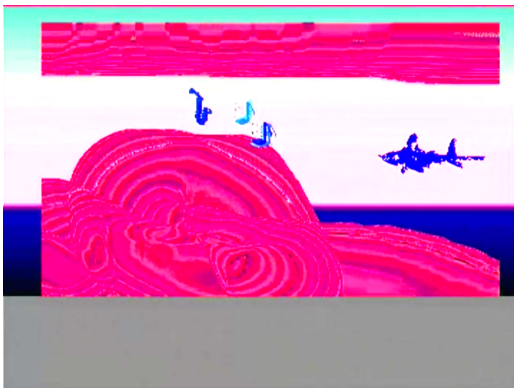

Figure 2: *screenshot of 2D scrolling game implementation*

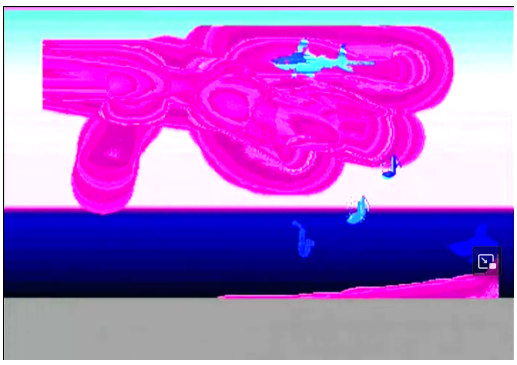

Figure 3: *screenshot of 2D scrolling game implementation*

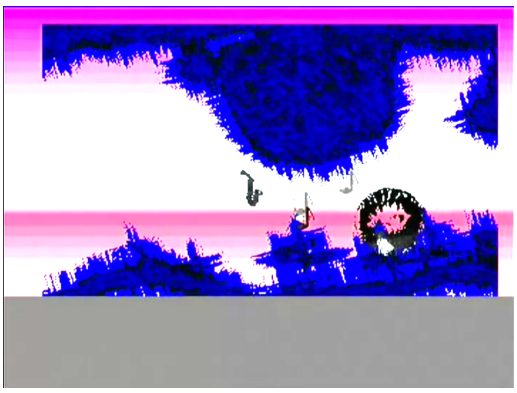

Figure 4: *screenshot of 2D scrolling game implementation*

Figure 5: *screenshot of 2D scrolling game implementation*

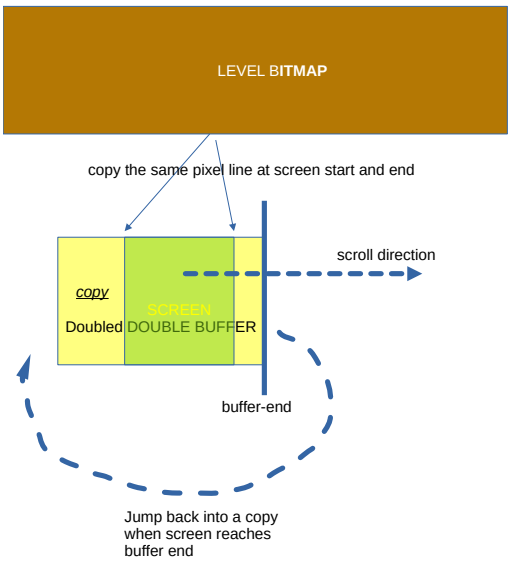

Figure 6: *Illustration of memory efficient single directional 2D scrolling*

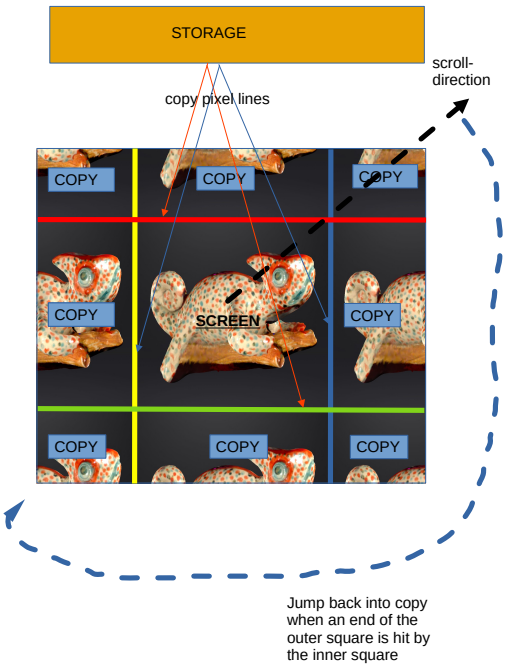

Figure 7: *Illustration of imaginary memory efficient multi directional 2D scrolling*

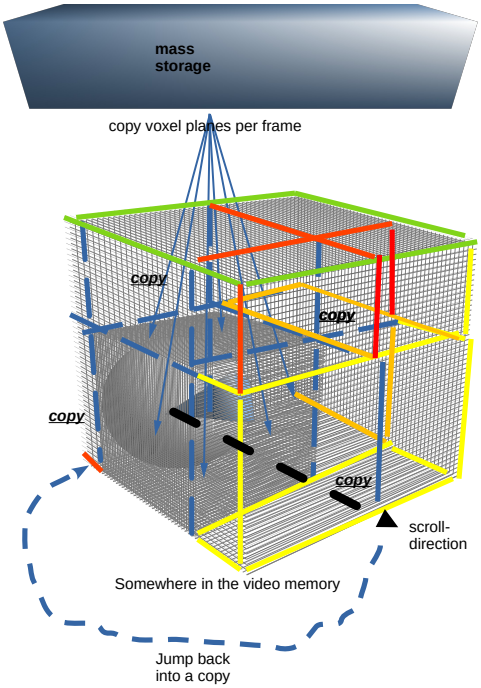

Figure 8: *Illustration of the proposed 3D algorithm version without multi-resolution nesting.*

2.2. Thinking About Memory Efficient Multi Directional Scrolling in 2D

Figure 7 illustrates memory efficient multi directional scrolling which was not implemented by us yet. Probably you can imagine that the same horizontal pixel lines would have to be copied at the red and green lines and the same vertical pixel lines at the yellow and blue lines from the here not shown much bigger level bitmap - but somewhat repeated. All in all it must be made sure that the here quadrupled double buffer always contains appropriate screen copies beside the window which is shown on the screen so that when the ends of the buffer are hit the screen window can jump back into a copy of the screen.

2.3. Voxel-Data-Scrolling or -Caching Proposal in 3D

Nowadays 3D-realtime-ray-tracing-visualisations of segmentation volumes have become possible on on consumer grade gaming hardware ([WPD24],[Mus13] ...). Unfortunately due to the cubic scaling of 3D voxel data these applications need a lot of video memory ([SHS24]). At the same time the difference between video memory size and mass storage memory size is still in the order of a factor of more or less than about 100 on most consumer systems. So to overcome the limited amount of video memory, some caching or prefetching between mass storage and video RAM would be feasible to render larger scenes. As proposed in this paper this could perhaps (not implemented by us yet) also be done using the key idea of the scrolling-algorithm from above but now in 3 dimensions instead of 2. Figure 8 illustrates the problem.

- The pyramid could symbolize the maximum sight of the ray-tracing/marching camera. It can be rotated freely.
- The maximum possible sights with arbitrary rotation is represented by the sphere.
- The outer cube could symbolize a 3D voxel array in video memory and is equivalent to the doubled/quadrupled double buffer of the 2D illustrations.
- The outer cube should be twice as large as the inner cube (in every scrollable dimension) as in the 2D equivalents.

The key idea in 3D is that whenever the inner viewing cube hits a boundary of the outer buffering cube the camera can be jumped back by half of the dimension of the outer cube since the voxels of the mass storage have been loaded at the planes intersecting the faces of the inner cube so that the 3D game player/user doesn't recognize the jump. The exact mechanism of updating the outer voxel cube on the planes intersecting the faces of the inner cube is hard to describe without implementation and testing, but it should work analog to the 2D versions described above. And it seems to be clear that the maximum viewing/ray-marching (tracing) distance should be set to half of the dimensions of the inner cube.

3. Evaluation

We only evaluated the single directional 2D version of the algorithm. The 3D version is only proposed and not implemented yet.

The 2D game was run and tested in amiga emulators and on real amiga500 machines. Both tests showed that the algorithm runs fast with 50 fps if few other game action is on the screen or 25 fps if too much other game action is on the screen. Slower frame rates were not recognized. This can be observed by playing the game. Hardware setup was a stock amiga500 machine with an expansion card for the trap door slot featuring 0,5 MB of "fast" RAM. The kickstart ROM version 1.3 was used. The monitor used was connected with a SCART to HDMI adapter. As monitor we used a BENQ monitor and other monitors like a 15khz capable NEC multi-sync monitor connected over a DSUB-VGA-adapter.

4. Related Work

There was published a new book in 2025 ([Cop25]) which also deals with such a game and contains also the single directional 2D version of the scrolling-algorithm. But it was implemented in assembler and uses tiles. Our implementation is mostly done in C [Oos19] (and some inline assembler [Oos21]) uses pixel lines and operates on hand crafted levels instead of tile maps. And the transfer to the multi directional 3D proposal is not contained in this book. The book does contain assembly source for multi directional scrolling too but - at the time of writing - probably without memory efficient scrolling of large levels and also only 2D. Related to our work might also be papers dealing with voxel data compression on the GPU because of limited video RAM ([PD24],[WPD24],[PKD]). It seems we have another approach than these papers because we don't use mainly compression and explain the caching in a mainly geometry-based way. Compared to [Mus13] our description is not thought yet for dynamic topology. But the 2D implementation also features some dynamic objects which might be possible in 3D in future too without loosing the ability to deal with large scenes. Furthermore there might exist some closed source implementations of amiga games that implement not only the single directional memory efficient scrolling but also memory efficient scrolling in multi directions but they mostly also use tiles and are not 3D. Finally there might also exist voxel-based 3D games but we don't know one to feature voxel scenes which exceed the memory limits of modern GPUs and real time ray-tracing.

5. Discussion

5.1. Maximum View Distance

Additional multi-resolution nesting could be used to extend the maximum distance of the sight by a factor of 2 for each new nesting level. If this is really necessary depends on the applications demands and on the speed tests that might be done in future after it is implemented.

5.2. Empty Space Skipping

The additional multi-resolution nesting mentioned above would also allow the renderer to implement empty space skipping since this nesting could also be thought of as an acceleration data structure with some memory redundancy. This redundancy does not affect the ray-marching speed but allows for easy caching of each nesting level as described above and shown in 8.

5.3. Adapting to the Scrolling Speed

Above we implicitly assumed that the scrolling speed would be at a constant rate of 1 pixel or voxel per frame. There might be applications that demand other scrolling speeds. In this case in the 2D version there would have to be copied not lines of pixels each frame but rectangular blocks of pixels with a thickness depending on the scrolling speed. In the 3D version there would have to be copied rectangular bricks instead of planes with a thickness depending on the scrolling speed. If the copied data itself constists of voxel cubes/bricks (as supposed later) further considerations might be taken into account which also depend on the scrolling speed as you can imagine.

5.4. Using Compression

Our approach as described so far might seem to be generally less suitable for GPU-based (de-)compression than other approaches ([PD24],[PKD],[WPD24]) since the data from which the decompression might happen is constantly updated whenever the camera scrolls (and also when you think games with additional dynamic interactions). Most practicable would perhaps be to decompress the whole volume on the mass storage before the application starts, as it might already be implemented in some implementations. Also hindering plane-based (de-)compression with CPU or GPU might be the fact that if more than one scrollable dimension is used the same data would have to be decompressed as if it was seen from different dimensions. This could perhaps be avoided if planes of voxel cubes/bricks would be copied instead of planes of single voxels in 8 . In this case it might be of advantage not to copy exactly one (repeated) plane of voxel cubes each frame but an amount of voxel cubes proportional to the scrolling speed in every scrollable dimension.

5.5. Using Pocedural Voxel Scenes (or Brick Maps or Hashed Voxels)

Also procedural voxel scenes (or maps constisting of bricks or other (re-)voxelizable things like hashed voxels [NZIS13]) could be used with this algorithm. The usage of the mass storage might not be necessary then. Instead the voxels on the planes intersecting the faces of the inner cube of figure 8 would not be copied but synthesized whenever the screen scrolls. Compared to on-the-fly-synthesis during ray marching this might also be faster. But some dynamic effects might require on-the-fly-synthesis or even more calculations (additionally). If brick maps or voxel cube maps are used the above metioned advantage of copying an amount of the voxel cube/bricks per dimension depending on the scrolling speed in every scrollable dimension per frame applies.

5.6. Using it for Light-Fields

Finally also light field or similar ([Lev96],[KDSD24]) data might be scrolled/cached with the algorithm in the future. In this case the empty space skipping would not be obligatory anymore since the empty space would be filled with light data and no ray marching would have to be done anymore (for static scenes). So it might also render faster but it would most likely be less suitable for dynamic (light) effects or objects.

6. Conclusion

As the 3D version is not implemented and approved (by us) yet, there is still much left to do. But the algorithm might still be a promising idea or tool also for the future since it might still solve also some 3D problems in a geometrically coherent and practicable way.

References

[Cop25] COPPI S.: *Amiga Assembly Game Programming*. Printed by Amazon Italia Logistica S.r.l. Torrazza Piemonte (TO), Italy, 2025. 1, 4

[KDSD24] KANDLBINDER L., DITTEBRANDT A., SCHIPEK A., DACHSBACHER C.: Optimizing Path Termination for Radiance Caching Through Explicit Variance Trading. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 7*, 3 (2024). doi:10.1145/3675381. 4

[Lev96] LEVOY M.: Light field rendering. *ACM Trans. Graph. 15*, 3 (1996), 289–302. 4

[Mus13] MUSETH K.: Vdb: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics 32* (June 2013), 22 pages. Presented at SIGGRAPH 2013. doi:10.1145/2487228.2487235. 3, 4

[NZIS13] NIESSNER M., ZOLLHÖFER M., IZADI S., STAMMINGER M.: Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (TOG)* (2013). 4

[Oos19] OOSTERKAMP E.: *Classic AmigaOS Programming*. Printed in Poland by Amazon Fullfillment Poland Sp. z o.o., Wroclaw, 2019. 4

[Oos21] OOSTERKAMP E.: *Bare-Metal Amiga Programming*. Printed in Poland by Amazon Fullfillment Poland Sp. z o.o., Wroclaw, 2021. 1, 4

[PD24] PIOCHOWIAK M., DACHSBACHER C.: Fast compressed segmentation volumes for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics 30*, 1 (2024), 12–22. doi:10.1109/TVCG.2023.3326573. 4

[PKD] PIOCHOWIAK M., KURPICZ F., DACHSBACHER C.: Random access segmentation volume compression for interactive volume rendering. *Computer Graphics Forum n/a*, n/a, e70116. doi:https://doi.org/10.1111/cgf.70116. 4

[SHS24] STADTER L., HOFMANN N., STAMMINGER M.: Neural Volumetric Level of Detail for Path Tracing. In *VMV2024* (2024), The Eurographics Association. doi:10.2312/vmv.20241197. 3

[WPD24] WERNER M., PIOCHOWIAK M., DACHSBACHER C.: SVDAG Compression for Segmentation Volume Path Tracing. In *Vision, Modeling, and Visualization* (2024), Linsen L., Thies J., (Eds.), The Eurographics Association. doi:10.2312/vmv.20241196. 3, 4