Level-of-Detail Algorithms for Outdoor Urban Game Scenes

Introduction

The level-of-detail (LOD) technique is an important performance optimization method in real-time rendering. Most game engines, including Unity, Unreal, and CryEngine, supports multiple LOD models for each game asset. Urban objects, such as buildings, vehicles, roads, and props such as traffic lights, are a very common type of game assets. Extensive research has been done in designing and implementing mesh simplification algorithms,[1–5] while they mainly focused on processing highly tessellated models generated from photogrammetry or other 3D reconstruction methods. On the contrary, models for game assets are created from scratch by 3D artists and may behave differently from tessellated objects under mesh simplification. Therefore, modifications to regular mesh simplification algorithms are necessary to reliably create LOD meshes for game assets. Compared with tessellated mesh, the mesh for a game asset usually include finer features than the main part of the model, for example, chairs on the courtyard of a house and HVAC equipment on the rooftop of an office building. When creating models with lower LOD, these finer features also need to be removed apart from simplifying the main model. Grid-based simplification methods, such as vertex clustering,[1] are not suited for this task, as the finer features may span multiple grids and are not guaranteed to be removed. Besides, game asset models are typically created from multiple geometric primitives, thus are highly non-manifold. Therefore, the **quadric error metrics** algorithm was selected as the basis of LOD algorithm implemented in this work. Two types of heuristics, surface area heuristic and boundary constraint, were incorporated into the algorithm to adapt to characteristics of game assets for urban objects. The LOD algorithm was implemented as a Python program and was evaluated on a diverse collection of urban objects.

Dataset

Specifically, the game assets investigated herein were exported from *Cities: Skylines*, which is the most popular city-building game in the past 5 years with more than 11 million copies sold. A key advantage of *Cities: Skylines* over other games involving urban objects is that it provides extensive plug-in support and tremendous amounts of community-contributed game assets. For example, more than 30,000 building objects and more than 10,000 vehicle objects are available for players to download from the Steam Workshop.[6] *Cities: Skylines* is based on the Unity game engine and utilizes two levels of detail, namely the “main mesh” and the “LOD mesh”. We will regard the LOD mesh created by the 3D artist as the ground truth of the mesh simplification result. Figure 1 shows the in-game screenshots for the main and ground truth LOD meshes for representative items in the dataset. A detailed description of the dataset is included in Appendix.

Methods

Games are real-time rendering systems and generate frames by forward or deferred rendering with geometry rasterization. Therefore, meshes for game assets are optimized for real-time rendering and have different characteristics from tessellated meshes, which directly resulted in the employment of the heuristics discussed herein.

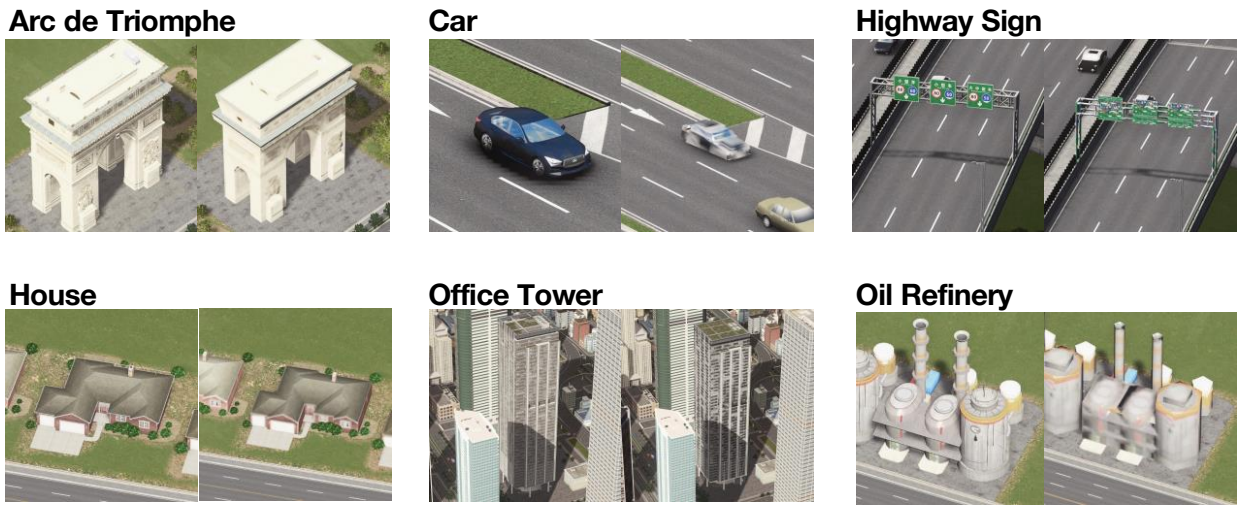

Figure 1. The urban objects dataset. The base mesh is shown in the left image and LOD mesh shown in the right.

Surface area heuristic

Unlike ray-tracing algorithms in which ray-triangle intersection (w.l.o.g, assume all meshes are triangulated) consumes significant computational cost, the performance of real-time rendering is more strongly dependent on the number of vertices and triangles in the model. Therefore, a planar surface, regardless of its orientation, should be subdivided as little as possible. This gives large variations in the surface areas of triangles in the game asset, whereas surface areas in tessellated meshes, such as the Stanford Bunny, are much less varied. The triangles with larger surface area are usually more important, and those with smaller areas are likely belong to the finer features and need to be removed first. However, the regular quadric error algorithm assumes equal weight on each triangle to which a vertex is connected. Therefore, a surface area heuristic was implemented as weighting both the fundamental quadric for each plane and the cost for each vertex by the surface area of triangles:

$$\boldsymbol{K_p} = S_k \boldsymbol{p}\boldsymbol{p}^T, \;\; \Delta(\boldsymbol{v}) = \left(\sum_{v \in k} S_k \right) \boldsymbol{v}^T \boldsymbol{Q} \boldsymbol{v}$$

The heuristic will encourage vertices with smaller total surface areas to be removed first and make the optimal vertex position of pair decimation closer to the larger triangles. Note that $\boldsymbol{Q}$ is already proportional S due to how quadrics are calculated, so the cost function has a square dependence on the surface area, which will almost guarantee that the algorithm process smaller triangles before larger ones.

Boundary constraints

Forward rendering systems shade all geometries in the field of view and then performs visibility calculations using hidden surface removal. Therefore, it is a good practice to remove all invisible triangles in the scene for game assets, for example, the bottom surface of a building or a car where it touches the ground. In addition, not all surface intersections in the mesh are covered by vertices and edges, since the model is constructed from geometric primitives. These two factors combined result in very ill-defined topology for game asset meshes, containing many separate manifolds and non-enclosed surfaces (Figure 2).

In the original quadric error metrics paper,[3] boundary constraints for meshes such as a terrain surface were applied by manually selecting the boundary edges. However, such approach could be intractable for game models because the prevalence of loose geometries. Hence, the boundary vertices, edges, and faces were automatically determined during the initialization of quadric errors. A very simple rule was used to determine boundary conditions: if a vertex has more neighboring edges than its neighboring faces, then it is a boundary vertex. Then, normal of constraint planes were calculated as the cross product of the edge connecting two boundary vertices and the normal of the face containing that edge. A special case exists where a non-boundary edge between two boundary vertices creating unnecessary constraints (consider the diagonal edge of one triangulated rectangle), but these cases are very rare in game models and did not significantly impact the simplification result. With the boundary constraints, the initial error quadric for a vertex is calculated as

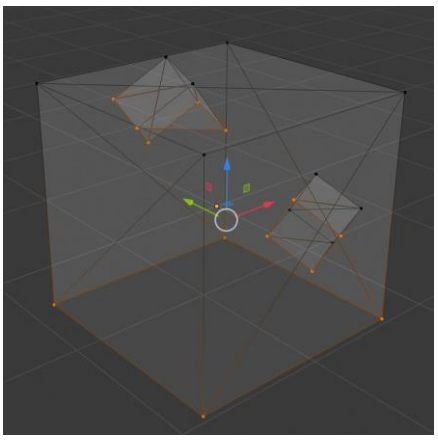

Figure 2. An example object modeled using 3 cubes with invisible surfaces removed when put on the ground. The object has 23 vertices while 13 vertices are on the geometry boundaries.

$$\boldsymbol{Q_V} = \sum_{v \in \boldsymbol{p}} \boldsymbol{K_p} + \sum_{\boldsymbol{c}} \boldsymbol{K_c} = \sum_{v \in \boldsymbol{p}} S_k \boldsymbol{p}\boldsymbol{p}^T + b \max\{S_k\} \sum_{\boldsymbol{c}} \boldsymbol{c}\boldsymbol{c}^T$$

where $\boldsymbol{p}$ is standard coefficient vector of the kth triangle and $\boldsymbol{c}$ is the set of constraint planes of a boundary vertex. b is a user-specified parameter to control the strength of boundary constraint. The area weight on the cost function does not include constraint surfaces so boundary vertices on smaller manifolds can be still decimated first.

Results and Discussion

To confirm the quadric error algorithm was implemented correctly, the LOD program was tested on the Stanford Bunny with all heuristics turned off. The decimated mesh with about 100 triangles generated from the original mesh with about 5000 triangles closely reproduced the result reported in Ref. 3 (see Appendix).

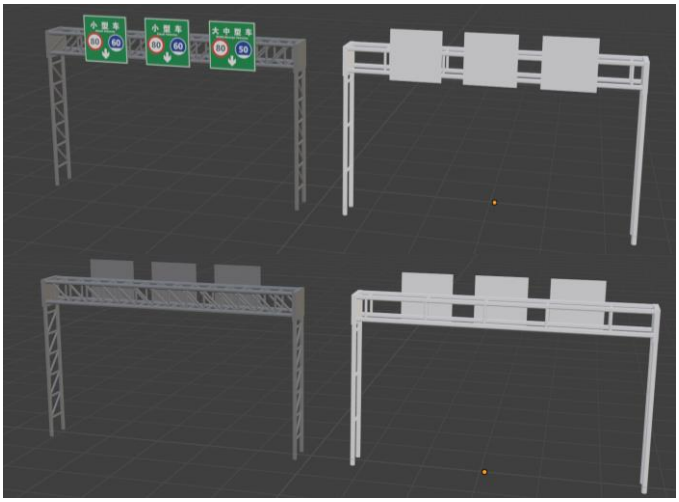

Figure 3. Front and back views of the overhead highway sign object with textured base mesh (3,100 tris) on the left and LOD mesh (640 tris) on the right.

The overhead highway sign object (sign.obj, Figure 3) was used to investigate the effect of surface area heuristic. Comparing the base mesh and the ground truth LOD mesh, the main difference between them is the removal of diagonal trusses. Each piece of truss is modeled as a hexagonal prism with Phong shading, thus removing them will dramatically reduce the complexity

of the model while preserving larger-scale features such as poles and sign boards. Figure 4 shows the simplification results for the base mesh using the original quadric error algorithm and the algorithm with surface area heuristic.

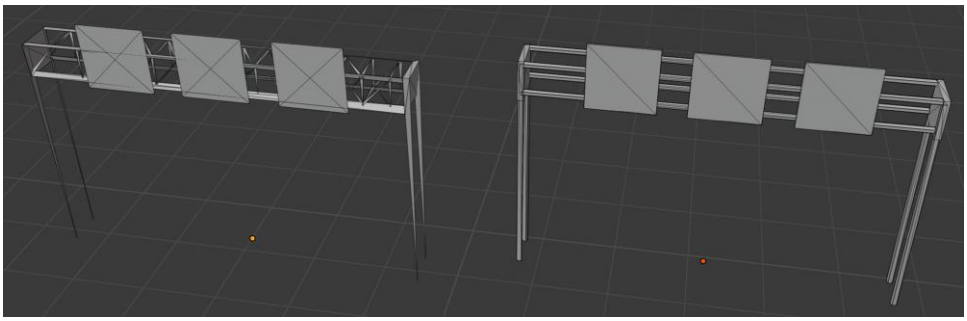

Figure 4. Front view of decimated meshes for the highway sign object using the quadric error algorithm (left, 200 tris) and with surface area heuristic (right, 204 tris)

The original quadric error algorithm gave a very poor simplification of the highway sign, with most of the trusses still present and poles reduced to elongated tetrahedrons or single triangles, which could be problematic when rendered from a far distance. This is because completely removing a manifold (decimating into a single edge) gives a very large cost for pair decimation, so it is strongly unfavored by the quadric error metric if the contribution of each triangle is equal. As a result, the algorithm tries to keep at least one triangle for each manifold. In contrast, removing the smaller features was successfully achieved with surface area heuristic, because the cost of complete removal was greatly reduced by the small surface areas. The simplified mesh using surface area heuristic even outperformed the ground truth LOD in preserving most of the structure using only 1/3 of the triangles.

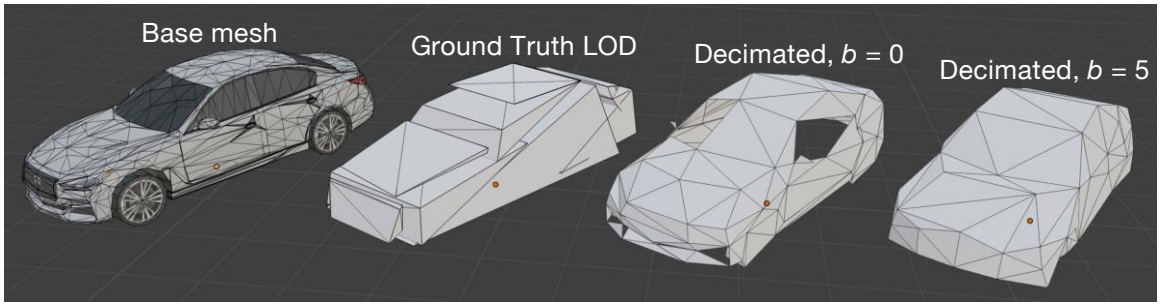

Figure 5. Front quarter views of the base mesh (3,145 tris) for the car object, ground truth LOD (76 tris), decimated mesh without boundary constraints (179 tris), and with boundary constraints (159 tris).

The car object (car.obj) was used to investigate the effect of boundary constraints (Figure 5). The base mesh of the object was likely a result from processing a much more complex mesh unsuitable for game rendering, therefore it also contains internal loose geometry apart from the removed bottom surfaces. For example, the black grill on the front bumper are modeled using triangles not connected to the body of the car. These loose geometries led to holes in the simplified mesh using the original quadric error algorithm, since boundaries are not preserved during decimation. This may be prevented using an extremely large pair threshold, yet both time and space costs will increase considerably as the number of vertex pairs approaches $O(N^2)$. In contrast, application of boundary constraints produced a better-behaved geometry without the presence of holes, since boundary vertices tend to stay on the boundaries which eases further merging into the main

geometry. The car object is fairly well tessellated, thus surface area heuristic was turned off by assigning $S_k = 1$ for all triangles.

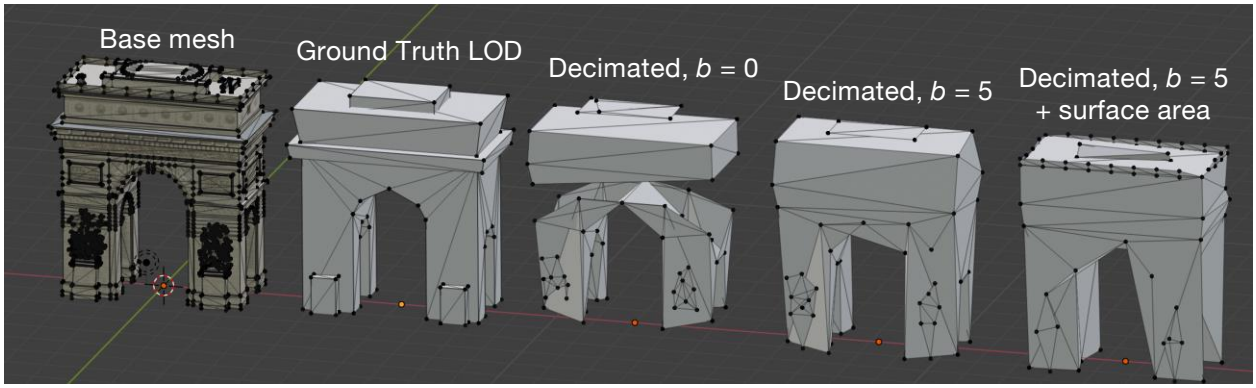

Figure 6. Front views of the base mesh (6,518 tris) for the Arc de Triomphe object, ground truth LOD (136 tris), decimated mesh without heuristics (129 tris), with boundary constraints (120 tris), and with boundary constraints and surface area heuristic (116 tris).

The modified quadric error algorithm was then tested on building objects, such as Arc de Triomphe. The ground truth LOD mesh did an excellent job in preserving the large-scale features using a minimum number of triangles, whereas the simplified mesh without any heuristic broke apart the geometry and left many vertices for the sculptures on the walls. Application of boundary constraints significantly improved the result, and surface area heuristics made the simplified mesh closer to regular geometries built using primitives. Interestingly, the rooftop fences are still present in the result with both heuristics, which will likely be removed with a finer adjustment of number of decimation steps.

Conclusion

A modified mesh simplification algorithm based on quadric error metrics was designed and implemented to optimize the quality on generate LOD models for game objects for urban scenes. Compared with the original quadric error algorithm, the modified algorithm achieved better results in preserve the large-scale features of the object and generates simplified meshes closer to what a 3D artist creates from scratch. Further improvements can be done on the algorithm such as applying edge volume weights on pair decimation to make the geometry more aligned to axes and enforcing symmetry in the mesh.

Submission Files

The files and directories included the submission are listed below:

- quadric_lod/: the directory containing the Python 3 program implementing mesh simplification. A readme file is available for usage instructions and required packages.
- models/: the directory containing OBJ models for the urban objects dataset.
- results/: mesh simplification results for the objects using original and modified quadric error metrics. The models are recorded every 25 or 50 decimation steps.
- video.mp4: video documentation for the program illustrating a complete process of creating an LOD mesh from an object in *Cities: Skylines*.

References

1. J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics: Methods and Applications*, pages 455–465, 199 3.
2. W. J. Schroeder, J. A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proc.)*, 26:65–70, 1992.
3. M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques (SIGGRAPH '97)*. 1997.
4. A. D. Kalvin and R. H. Taylor, Superfaces: polygonal mesh simplification with bounded error, *IEEE Computer Graphics and Applications*, 16:64-77, 1996.
5. M. Isenburg, P. Lindstrom, S. Gumhold and J. Snoeyink, Large mesh simplification using processing sequences in *IEEE Visualization, 2003. VIS 2003.*, Seattle, WA, USA, pp. 465-472, 2003.
6. Cities: Skylines – Steam Community. https://steamcommunity.com/app/255710/workshop/ (accessed Apr 23, 2020)

Appendix

Table S1. Game assets in the urban objects dataset. The models shown in Blender viewport is mirrored from in-game scenes because Unity uses a left-handed coordinate system.

File name	Description	Source
arc.obj	Arc de Triomphe, Paris, France	Included in *Cities: Skylines* base game
car.obj	2018 Infiniti Q50, 4 door sedan	https://steamcommunity.com/sharedfiles/filedetails/?id=1186381629
condo.obj	Modern high-rise residential tower	https://steamcommunity.com/sharedfiles/filedetails/?id=1860794227
house	Single-family housing with garage and driveway	https://steamcommunity.com/sharedfiles/filedetails/?id=648215850
oil.obj	Oil refinery	Included in *Cities: Skylines* base game
sign.obj	Overhead highway sign	https://steamcommunity.com/sharedfiles/filedetails/?id=2048848384
suv.obj	2020 GMC Yukon Denali, SUV	https://steamcommunity.com/sharedfiles/filedetails/?id= 2034603006
terminal.obj	Grand Central Terminal, New York, USA	Included in *Cities: Skylines* base game
tower.obj	111 South Wacker Drive, Chicago, USA	https://steamcommunity.com/sharedfiles/filedetails/?id= 1593207311

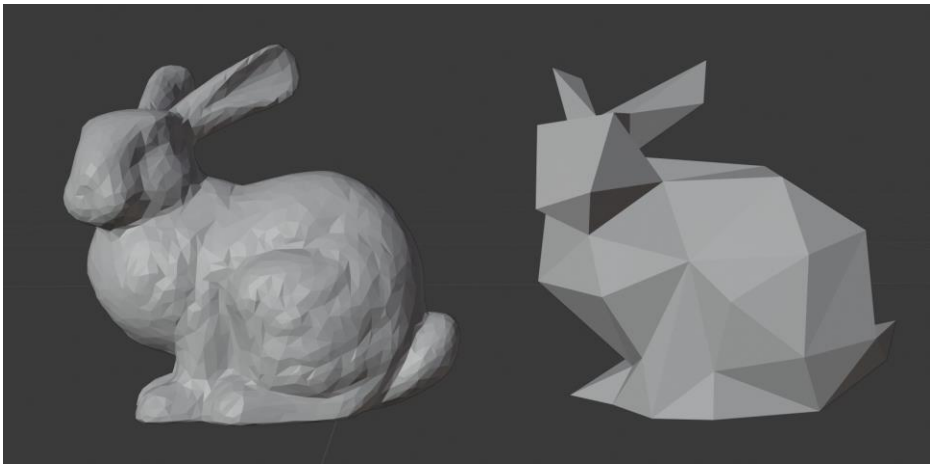

Figure S1. Original (left) and decimated (right) meshes of Stanford Bunny.