A Framework for OpenGL Client-Server Rendering

Christopher Dyken, Kjetil Olsen Lye, Johan Seland,
Erik W. Bjønnes, Jon Hjelmervik, Jens Olav Nygaard, and Trond Runar Hagen
SINTEF ICT
Oslo, Norway

***Abstract*—We present a software framework that facilitates the development of OpenGL applications utilizing the limited GPU capacities of a portable client in combination with the high-end rendering hardware on a server. The resulting web-application uses standard technologies and can be run on a wide variety of devices, such as smart phones, tablets and laptops. The framework is designed so that it is simple to make an existing OpenGL application into a web-application, gradually adding client-side rendering. Furthermore, it provides automatic network scaling to provide interactivity even on poor connections.**

I. Introduction

The computing landscape has seen massive shifts in the last decade. As predicted by Moore's Law [1], transistor density is still doubled every 18 months, allowing us to balance power consumption and performance when designing chips. Therefore we can build low-power, portable devices as well as parallel, high-performance servers. A powerful combination of these is the use of a portable client to access remote data, processed on servers, often called *cloud computing*.

The combination of client-server rendering is particularly powerful for scientific visualization, as it remedies several challenges often encountered by such applications. The datasets are often very large, making them unsuitable both to transmit and store on the client. Furthermore they can have copyright and intellectual property issues, making it infeasible to transmit the raw dataset. Finally, state-of-the art rendering algorithms often rely on recent GPU hardware with specific driver requirements, which can be hard to satisfy on portable devices.

While there exist several approaches that can display images from servers on a client, they often have latency issues and can be difficult to adapt to in-house applications. With most of todays portable clients being equipped with GPUs, and WebGL standardizing web-based rendering, new approaches become viable.

We propose a hybrid solution, utilizing both the limited GPU of the client device, and the powerful server-side GPUs. The main rendering is done on the server, allowing the application access to traditional GPUs, where it can make use of large data sets and advanced rendering techniques. The client device is responsible for GUI and user interactions, rendering a simplified geometry called *proxy geometry* see Figure 1 for example. This can be navigated in, and possibly manipulated, by the users who instantly sees the results of their actions on the device. The latency of receiving the resultant high-quality rendered image from the server, based on the users input, is thus hidden, negating the problem of slow or congested networks.

This divison of work also lends itself to porting existing applications, as mostly the GUI needs to be adapted to the framework, with only minor changes required to the main application and OpenGL renderer.

In this paper we first give a quick overview of various related approaches to networked graphics and rendering, before presenting Tinia, our main contribution, giving both a technical overview and some more details about the rendering. We then briefly discuss how existing applications can be integrated, before our concluding remarks.

A. Related Work

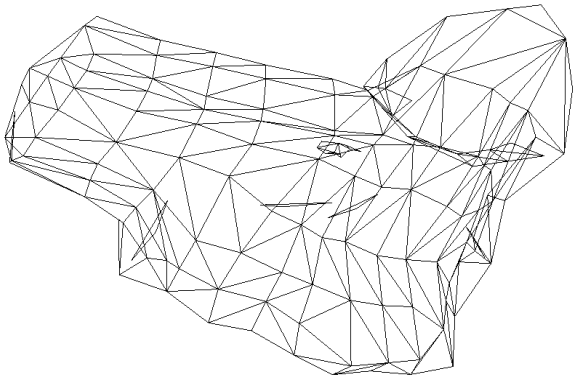
Over the recent decades it has been many different approaches to serving graphics over networks. X-terminals [2] were popular in the 1990s and used client side graphics hardware to accelerate mouse and window movements. Later various iterations of cheap, thin-clients provided local GUI rendering, but relied on servers for heavy computations. This trend is continued to the present day, sometimes dubbed the *post-PC era* where much computing happens on smartphones and tablets.

Several applications has been extended to allow remote rendering. Noguera et al. [3] describe navigating large terrain on mobile devices where the terrain closest to the viewer is rendered on the client. ParaViewWeb [4] is a web interface for ParaView enabling remote visualizations of simulation data. Similarily, Niebling [5] describes how a specific application, COVISE, has been extended with a WebGL renderer and long polling based GUI integration, techniques also used by our framework.

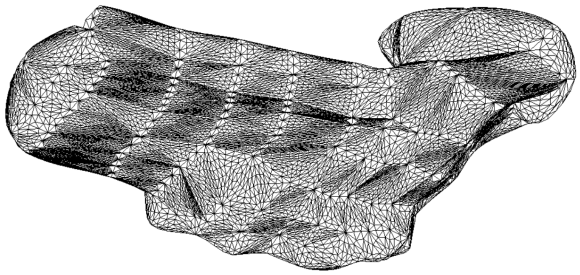
Anttonen [6] lists several popular WebGL frameworks used for games and scientific visualization. However, none of them allows for the use of a server back-end to increase rendering fidelity.

Recently high-bandwith networks has proliferated to the extent that several companies, such as Onlive [7] and Gaikai [8], now offer gaming-as-service through a video streaming solution. Common to these platforms is that high-end games are rendered on servers and the resulting video images are streamed in real-time over the Internet.

For a more thorough investigation of further approaches to rendering and visualization as a service see [9].



(a) A proxy geometry.



(b) A highly tessellated geometry

Fig. 1: An example of a lightweight client geometry versus a dense server geometry. In both cases the geometry represents a gall bladder for use in surgical simulations.

II. The Tinia Framework

Tinia is a software framework written in C++ and JavaScript, released under the GNU Affero GPL version 3. The framework supports creating ordinary desktop applications as well as client-server applications with OpenGL-rendering. The framework is designed so it is possible to have the same API for both of theses modes of operation. We will only describe the creation and anatomy of a client-server application in Tinia.

Today's cloud-platforms, such as Amazon EC2, supports GPGPU computing as-a-service, but not yet OpenGL rendering. However, better virtualization support on GPUs is is becoming available [10], so we expect is should be possible to deploy Tinia applications on such services in the near future.

Conceptually the Tinia framework provides a set of *content streams* between the client and the server, and a suite of *messaging components* that transports the streams over the network.

A. Content Streams

The content streams have different bandwidth and latency requirements, and they are therefore encoded using different technologies.

- *Server Rendering* is the result of the server-side OpenGL commands, which is passed into Tinia as an OpenGL FrameBuffer Object (FBO). Since the framework only needs the final image, it is agnostic to complex, multi-pass rendering algorithms.
- *Client Rendering* is a set of WebGL commands that are built on the server, before they are streamed and executed on the client.
- *Model* is a description of the user interface to be displayed in the web browser and commonly used variables such as, the OpenGL matrices, viewpoint etc., and other application defined parameters. The client always holds a full version of the model, and changes are communicated between the server and client. The name is due to the Model-View-Controller design pattern [11], and must not be confused with a geometric model. It is through the model that the server and client communicates user input.

B. Messaging Components

Conceptually, the framework consists of four different messaging components with dedicated responsibilities, that encodes, decodes and transports the rendering stream over the network. The message paths and encoding technologies is illustrated in Figure 2.

- *Job* is the application logic, possibly consisting of an existing OpenGL engine
- *IPC-Controller* handles communication between the job and the web server module, as well as grabbing the resultant OpenGL FBO.
- *Web server module* is an Apache 2 module which generates a web page sent to the browser.
- *web-application* provides the user interface and communicates with the server through XML. It both decodes/encodes and interprets various content streams.

The web-application communicates with the web server over HTTP. In turn, the web server uses POSIX shared memory to message the IPC-Controller, which again resides in the same memory address space as the job and shares the same C++ objects.

C. The Event Loop

The Tinia event loops allow for changes to the model to happen on both the client and the server. Client side changes are typically GUI events, such as changing the viewpoint, while server initiated changes might be data updated from a running simulation. These changes must be propagated to the other endpoint.

However, since both ends hold a full version of the model at any time, only the updated variables are communicated. To facilitate this, the model class implements the observer design pattern: If something triggers a change in the model, a set of listeners are notified. See Figure 3 for an illustration.

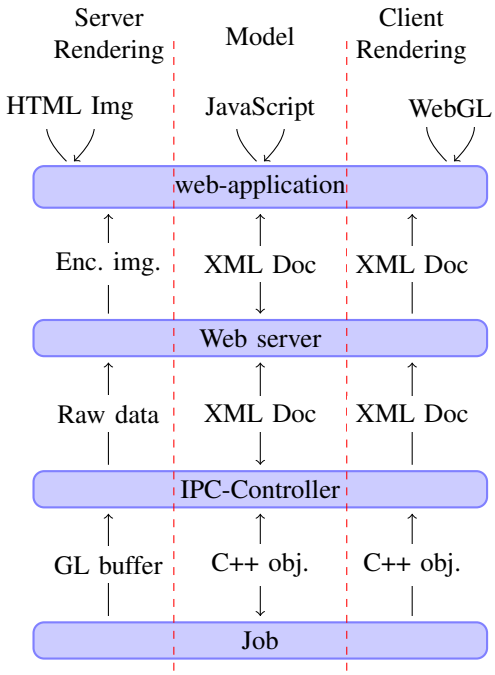


Fig. 2: Component communication diagram, illustrating how each component encodes and decodes the various content streams.

On the client these can be other user interface widgets or programmer defined scripts, and on server they can be any C++ class which relates to the model. One important listener on both endpoints is the model sender, which communicates changes back to the other end.

To ensure that the other end receives all changes in order, the model sender only allows one transfer at a time. If it receives a notification during transmission, it will enqueue the need for another transmission, which will be sent once the current transmission is completed. One transmission may contain the values of several variables, but only one value per variable in the model. Thus the other endpoint is not notified of every change in the model, just the final value at the time of transmission.

To avoid loops in the update handling, the model receiver signals the model sender that a change is imminent, thereby disabling the transmission to the other endpoint.

The transmission policy described above automatically implements a form of latency scaling, ensuring that the application does not saturate the network. For instance, changes in the viewpoint may happen several times per transmission, but only when there is an open slot is the viewpoint communicated to the server. This also greatly reduces the workload on the server, as it requires far fewer redraws of the scene.

The realization of the event loop, using widely different technologies on the two endpoints, are described in the coming sections.

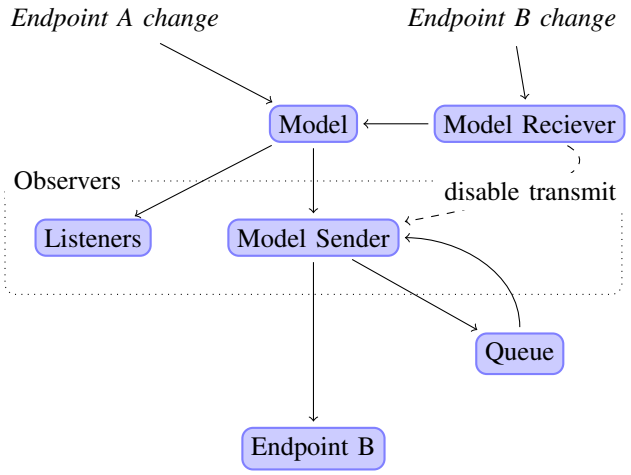


Fig. 3: The Event Loop of a Tinia Application. Conceptually the same event loop is executed both on the client and the server.

D. The Web-Application

The web-application is a generic application implemented in JavaScript using HTML5 and WebGL. Because of this, it is possible to load the application directly without relying on additional software on most modern browsers. During startup the application receives an XML schema describing the current job's model and the initial state of the model. From this the application automatically generates a user interface and starts the event loop described above. In addition to this behaviour, the programmer may optionally add other scripts to the web-application. This enables much of the application logic to take place on the client side, thereby reducing both network traffic and latency. See Figure 4 for a screenshot of a generated GUI.

All communication between the server and client is handled through asynchronous HTTP requests. Responses from the server can be acquired in two ways: through events initiated in the user interface or events initiated by the server communicated to the client through *long polling*, described in [12]. Using this polling method, the server holds the request open even if it has no data to send at the moment. Once information is available it is immediately sent. The client will then immediately open a new request, so that the server always has an available request open.

E. Viewer controls handling

To facilitate different view controls the application programmer may write JavaScript classes, called *viewlets*, that respond to mouse and keyboard events, and executed fully on the client. Some common predefined controls come bundled with the framework.

On initialization the framework instructs each viewlet with which variables in the model it should update, typically the viewpoint and OpenGL-matrices, but the application is free to define other ways to transfer the view-state. As the viewlets lives entirely on the client, there is no need for the web-application to query the server for new state. The server-side

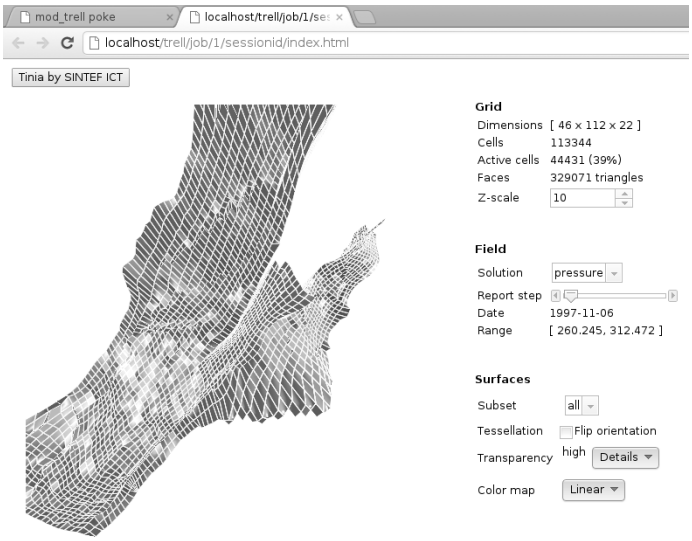


Fig. 4: A reservoir visualization web-application, allowing for loading and navigating massive datasets on demand. The GUI is automatically generated by the framework.

application queries the model for the results of the viewlets and renders the scene accordingly.

It is possible to have several viewlets on one OpenGL scene, ie. one for handling picking, and multiple viewlets responsible for generating the OpenGL-matrices, allowing for multiple camera models.

F. The Server-Side Processes

The web-server process runs as an Apache module, thereby making the large library of existing Apache modules available for security and scalability. However, as the Apache web server, by design, frequently kills and spawns new module processes, the lifetime of a given process of the module is therefore much shorter than the typical lifetime of the server side process. This mandates the need for the IPC-Controller being a separate process.

The job is directly linked with the IPC-Controller creating a C++-program which is executed through the framework. Since each program is a separate process, all resource sharing is handled by the operating system, thus the framework is able to support several program instances on the same server. The GPU can also be shared by several instances, all managed by the operating system. At present the framework does not handle several GPUs for several process, and each process is assigned the same GPU. Techniques amending this could be added to a future version of Tinia.

When Apache receives a request from the web-application, it forwards the request to the Apache module, which in turn interprets the request and notifies the IPC-Controller via POSIX IPC. The controller handles the request, possibly by interacting with the job, and it transfers the result back to the controller via IPC.

III. OpenGL Rendering

The OpenGL umbrella now covers several implementations, WebGL for browsers, OpenGL ES (Embedded Systems) for smart phones etc., as well as traditional OpenGL. Below we describe how Tinia combines WebGL and regular OpenGL.

A. Client rendering

Conceptually the client rendering is a simple version of the server rendering, typically well suited for rendering on a thin client. The client rendering is subject to the same OpenGL matrices as the server rendering, and will typically mimic the server rendering as well as the client device allows.

Currently the client renders the proxy geometry whenever the user is interacting with the viewport or an image is loading from the server. This is done to hide latency and promote interactivity. This behaviour can be replaced with a smarter heuristic for determining when latency requires the use of proxy geometry instead of images from the server.

The client rendering is specified from the server application through a renderlist API. This API allows for specifying and storing the proxy geometry, as well as rendering algorithm and shader programs. WebGL-code is thereafter automatically generated from this by the framework. The renderlist may be altered by the application during its entire lifespan.

B. Server rendering

The server-side application typically holds all relevant rendering data and it is responsible for drawing a high-detailed version of the geometry. Any OpenGL-rendering techniques are allowed, as long as the final result is written to the FBO specified by the framework.

Since the framework is agnostic to what OpenGL calls are made, there are no hindrances to employing advanced rendering techniques in the server application. It is also fully possible to embed an existing rendering engine in a Tinia application.

When the IPC-Controller receives a request for a server-rendered image, it in turn invokes the render procedure in the job. Once the job completes its rendering, the IPC-Controller dumps the raw FBO data to a shared-memory area. The web-server then encodes the data to the image format requested by the web-application and forwards it to the web-application.

The image grabbing is quite light-weight and does not affect the performance of the render algorithm.

IV. Porting an existing application

Tinia is designed to facilitate integration of existing OpenGL applications. We assume that such applications typically have a relatively simple main loop, consisting of a render pass and event handling. To make such an application available through Tinia, three simple steps are mandatory and the last step is recommended.

1) Expose the variables you want the user to modify, by adding them to the model. The user-interface will be generated automatically, but can be manually specified.

2) Utilize the OpenGL information, such as matrices, that the framework provides.
3) Map events made by the framework into application's event handling.
4) *(Optional)* Define a proxy geometry and possibly client-side shaders. Without this step the framework will mimic the behaviour of video streaming solutions.

In our experience, the execution of these steps is relatively straightforward, and the framework has seen use in a variety of fields, such as reservoir visualization, surgical simulation and airport monitoring software.

V. Future Work

There are many extensions possible for the framework. One planned feature is the automatic generation of proxy-geometry from higher order surfaces such as spline patches, possibly using the hardware assisted tessellators available on discrete GPUs, functionality that is not yet exposed on devices running WebGL.

A possibility is to only send the alterations to the previous sent image in the form of change sets, utilizing the techniques found in [13]. Since the web browser supports various lossless image formats, the framework could encode the change set as a compressed image and use the browser for decompression. This could greatly reduce bandwidth requirements, and hence would provide a noticeable decrease in latency for configurations with low bandwidth. Meanwhile the web browser is able to possibly use special hardware for decoding the image. The actual merging of the change set and the previous image could be implemented as a WebGL program.

For dynamic scenes, the server application could provide a video stream for the web-application. While the framework can be used for this without video streams, it would require that the web-application continuously update the HTML image. As most mobile devices are equipped with video decoding hardware, using video streams could reduce battery drain for application with animation requirements, and at the same time benefit from the extra compression video encoding provides.

Another natural extension is to provide other client-side targets beside the web-application, natural targets are iOS and Android applications, possibly yielding higher performance.

VI. Conclusion

We have described our software framework for hybrid client-server rendering of OpenGL applications. The framework is already in use in several research applications developed by our partners and us. It has also been made available under an open-source license and can be downloaded from www.tinia.org.

Acknowledgment

The authors would like to thank our industrial partners Ceetron, SimSurgery and Statoil for invaluable discussions and feedback. This work was funded in part by NFR Grant 201447. The Norne reservoir data are courtesy of Statoil, and its license partners ENI and Petoro, coordinated by Center for Integrated Operations at NTNU. The gallbladder surface is courtesy of SimSurgery.

References

[1] G. Moore, "Progress in digital integrated electronics," in *Electron Devices Meeting, 1975 International*, vol. 21, 1975, pp. 11 – 13.
[2] L. Mui, E. Pearce, and O. . Associates, *X Window system administrator's guide: for X version 11*, ser. Definitive guides to the X Window System. O'Reilly & Associates, 1992.
[3] J. M. Noguera, R. J. Segura, C. J. Ogáyar, and R. Joan-Arinyo, "Navigating large terrains using commodity mobile devices," *Computers and Geosciences*, vol. 37, no. 9, pp. 1218 – 1233, 2011.
[4] "Paraviewweb website," http://paraviewweb.kitware.com/PW/, accessed August 2012.
[5] F. Niebling, A. Kopecki, and M. Becker, "Collaborative steering and post-processing of simulations on hpc resources: everyone, anytime, anywhere," in *Proceedings of the 15th International Conference on Web 3D Technology*, ser. Web3D '10. New York, NY, USA: ACM, 2010, pp. 101–108.
[6] M. Anttonen, A. Salminen, T. Mikkonen, and A. Taivalsaari, "Transforming the web into a real application platform: new technologies, emerging trends and missing pieces," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, pp. 800–807.
[7] "Onlive website," www.onlive.com, accessed August 2012.
[8] "Gaikai website," www.gaikai.com, accessed August 2012.
[9] C. Mouton, K. Sons, and I. Grimstead, "Collaborative visualization: current systems and future trends," in *Proceedings of the 16th International Conference on 3D Web Technology*, ser. Web3D '11. New York, NY, USA: ACM, 2011, pp. 101–110.
[10] "Nvidia cloud computing," http://www.nvidia.com/object/cloud-computing.html, acessed August 2012.
[11] E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
[12] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins, "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP," RFC 6202 (Informational), Internet Engineering Task Force, Apr. 2011.
[13] P.-P. Vázquez and M. Sbert, "Bandwidth reduction for remote navigation systems through view prediction and progressive transmission," *Future Generation Computer Systems*, vol. 20, no. 8, pp. 1251 – 1262, 2004, computer Graphics and Geometric Modeling.