Defence Research and Development Canada | Recherche et développement pour la défense Canada

DEFENCE DÉFENSE

Air Defence System:

Profiling and Optimizing the KARMA IRSG Module

M.-A. Labrie
E. Rouleau
M. Desmeules

Prepared By:
LTI software & engineering
825 Boul. Lebourgneuf, Bureau 204
Québec, Canada
G2J 0B9

Contractor's Document Number: LTI-ADS-2012-1
Contract Project Manager: Marc-André Labrie
PWGSC Contract Number: W7701-083373-AT10
CSA: Jean-Francois Lepage, Defence Scientist 418-844-4000 (4192)

Defence R&D Canada – Valcartier

Contract Report
DRDC Valcartier CR 2012-070
July 2012

Canada

Air Defence System:

Profiling and Optimizing the KARMA IRSG Module

M.-A. Labrie
E. Rouleau
M. Desmeules

Prepared By:
LTI software & engineering
825 Boul. Lebourgneuf, Bureau 204
Québec, Canada
G2J 0B9

Contractor's Document Number: LTI-ADS-2012-1
Contract Project Manager: Marc-André Labrie
PWGSC Contract Number: W7701-083373-AT10
CSA: Jean-Francois Lepage, Defence Scientist

Defence R&D Canada – Valcartier

Contract Report
DRDC Valcartier CR 2012-070
July 2012

Abstract

The main objectives of task AT-10, as part of contract W7701-083373, was to profile the performance of the infrared scene generator (IRSG) module of the KARMA simulation framework, pinpoint bottlenecks, optimize critical portions of code through basic corrective measures, and update the SMART (Suite for Multi-resolution Atmospheric Radiative Transmission) library to the latest version. The work has been carried out from January 2012 to March 2012. The profiling has been limited to a basic scenario using the IRSG and the wideband mode. The code optimizations allowed to double the frame rate for this scenario. Some optimization avenues were investigated and could be realized as a part of another task or contract.

Résumé

Les objectifs de la tache AT-10, du contrat W7701-083373, étaient d'effectuer le profilage des performances du module de génération de scène infrarouge (IRSG) de l'environnement de simulation KARMA, d'identifier les éléments critiques, d'optimiser certaines portions critiques par des mesures correctives de base, et de mettre a jour la librairie de calculs atmosphérique SMART (Suite for Multi-resolution Atmospheric Radiative Transmission). Les travaux ont été effectués de janvier 2012 à mars 2012. Le profilage des performances a été limité à un scénario simple utilisant l'IRSG en mode large-bande. Les optimisations au code ont permis de doubler la cadence pour un tel scénario. Des avenues ont aussi été identifiées pour une éventuelle deuxième phase d'optimisation.

This page intentionally left blank.

Executive summary

Air Defence System: Profiling and Optimizing the KARMA IRSG Module

M.-A. Labrie; E. Rouleau; M. Desmeules; DRDC Valcartier CR 2012-070; Defence R&D Canada – Valcartier; July 2012.

Introduction: The KARMA simulation framework is currently used at DRDC Valcartier to simulate engagement level scenarios involving air targets. Within the Virtual Proving Ground (VPG), the KARMA simulation framework and its Infrared Scene Generation (IRSG) module will also be used for main-in-the-loop and hardware-in-the-loop applications. To allow for these time critical simulations to be performed, it was necessary to increase the frame rate of the IRSG module. As a preliminary phase of this improvement, task AT-10 of contract W7701-083373 was intended to determine the potential for frame rate improvements with the infrared scene generator (IRSG) module of KARMA.

Results: The work, carried out from January 2012 to March 2012, allowed to profile the performance of the IRSG module, pinpoint bottlenecks, optimize critical portions of code through basic corrective measures, and update the SMART (Suite for Multi-resolution Atmospheric Radiative Transmission) library to the latest version. The basic code optimizations performed within the task allowed to double the frame rate.

Significance: The work has shown the potential of the IRSG module for frame rate improvements, and demonstrated that the IRSG module would be appropriate for the envisioned real-time applications with proper optimisations done. It also allowed to identify optimization avenues that could have an important impact on the performance of the IRSG module.

Future plans: The next step will be to perform the second phase of optimisation as a function of the profiling resulting from this task.

Sommaire

Air Defence System: Profiling and Optimizing the KARMA IRSG Module

M.-A. Labrie; E. Rouleau; M. Desmeules; DRDC Valcartier CR 2012-070; R & D pour la défense Canada – Valcartier; juillet 2012.

Introduction: L'environnement de simulation KARMA est actuellement utilisé par RDDC Valcartier pour simuler des scénarios de niveau engagement impliquant des cibles aériennes. Dans le cadre du polygone d'essai virtuel, l'environnement de simulation KARMA et son module de génération de scène infrarouge (IRSG) seront aussi utilisés pour des applications humain dans la boucle et matériel dans la boucle. Afin de permettre l'exécution de telles simulations, dont le temps d'exécution est critique, il était nécessaire d'améliorer la cadence d'affichage du module IRSG. En phase préliminaire, la tache AT-10 du contrat W7701-083373 visait à déterminer le potentiel d'amélioration des performances du module IRSG de KARMA

Résultats: Les travaux, effectués de janvier 2012 à mars 2012, ont permis d'effectuer le profilage des performances du module IRSG, d'identifier les éléments critiques, d'optimiser certaines portions critiques par des mesures correctives de base, et de mettre à jour la librairie de calculs atmosphérique SMART (Suite for Multi-resolution Atmospheric Radiative Transmission). Les optimisations de base effectuées au au code lors de cette tache ont permis de doubler la cadence d'affichage.

Importance: Les travaux ont démontré le potentiel d'amélioration des performances du module IRSG, et ont démontré que celui-ci pouvait être approprié pour les applications en temps réel envisagées, une fois certaines optimisations faites. Des avenues d'optimisation pouvant avoir un impact important sur les performances ont aussi été identifiées.

Perspectives: La prochaine étape sera d'effectuer la seconde phase d'optimisation en fonction du profilage des performances résultant de cette tache.

LTI

Informatique & génie
Software & engineering

Air Defence System

Profiling and Optimizing the KARMA IRSG Module

Marc-André Labrie - LTI
Eric Rouleau - LTI
Mathieu Desmeules - LTI

Mr. Jean-Francois Lepage - DRDC-Valcartier
W7701-083373-AT10

23 March 2012

This page intentionally left blank.

Table of Contents

List of Figures

List of Tables

1 Introduction

The main objectives of task AT-10, as part of contract W7701-083373, was to 1) profile the performance of the infrared scene generator (IRSG) module of the KARMA simulation framework, 2) pinpoint bottlenecks, and 3) optimize the critical part of the code through basic corrective measures.

Another objective was to update the SMART (*Suite for Multi-resolution Atmospheric Radiative Transmission*) library used by the KARMA framework to the newest version available. The work was carried out from January 2012 to March 2012.

This report focuses on presenting the results of the performance profiling (before and after the implementation of the optimizations). The optimizations are summarized as well as some ideas regarding optimization avenues that could have an important impact on the performance of the IRSG module.

2 General information

2.1 SMART

SMART is a C++ library developed at DRDC-Valcartier [1]. The main purpose of SMART is to calculate atmospheric values such as transmitted solar irradiance, atmospheric fluxes, path and background radiances, and transmittance. This library can produce outputs in both spectral or wideband correlated-k (CK) format.

This library takes an important place in the IRSG module, allowing obtaining precise atmospheric values. To take advantage of the various fixes and improvements done to the library since the last release, the choice was made to first update KARMA with the most recent version of SMART, before proceeding with the optimization phase. The last time this library was updated in KARMA was in June 2010 (revision 961). Therefore, the library was updated to revision 1088 at the beginning of this task. A final update was done prior the end of the task in March 2012 (revision 1089). Consequently, the verification and validation (V&V) report of the signature management capability was updated to take into account the new version of the library on the results produced by the IRSG module. The results, for the cases depicted in the V&V, are almost the same (very small modifications to some values) as before the library update.

KARMA uses the scene scattering mode (as opposed to sensor scattering mode) available with SMART. With this mode, the first call to *SMART::update* is very long but subsequent ones (made before each frame is generated) are very short. However, with the analysis of a first profiling session, all our calls to *SMART::update* were long. This was caused by an unusual utilization of SMART combined with the utilization of the SMART *onlyWide* mode. A refactoring of the *SmartAdapter* class (which allows using SMART within KARMA) associated with a modification in SMART has allowed to correct the encountered problem (see Table 1 and Table 2, produced with the scenario presented in section 2.2).

Table 1: SMART::update timings before SMART library update and SmartAdapter refactoring.

1st call to *SMART::update*	763 ms
Average time for other calls to *SMART::update*	140 ms

Table 2: SMART::update timings after SMART library update and SmartAdapter refactoring.

1st call to *SMART::update*	4168 ms
Average time for other calls to *SMART::update*	0.017 ms

The first call to *SMART::update* is very long but is done during the IRSG initialization phase, prior the simulation begins.

2.2 Baseline scenario

The performance profiling has been performed using a basic engagement scenario (a man-portable air-defense system (ManPADS) missile against a CC130 platform), without complex rendering mechanisms and 3D models, in order to focus on optimizing the scene generation process rather than the overall simulation process. The major parameters are presented in Table 3.

Table 3: Main parameters of the baseline scenario.

Antialiasing	None
AtmosphereSmart model	Base parameters except for the spectral band which is: 2.5 - 5.5 µm
Background	Uniform
Calculation mode	Wideband
Duration	3.00 seconds
Flares	Without
Frames generated	301
Image size	512 x 512 pixels
Platform	CC130 generic (*CC130_IR.flt*)
Platform 3D model characteristics	1206 polygons, no LODs
Platform Database	10 temperatures (*IRColor*) 4 materials (*IRMaterial*) 10 combinations temperature-material
Sensor spectral response	3.0 - 5.0 µm
Skybox	None
Terrain	None

The 3D model used for the CC130 platform is shown in Figure 1, in a scribed (solid and wireframe) appearance.

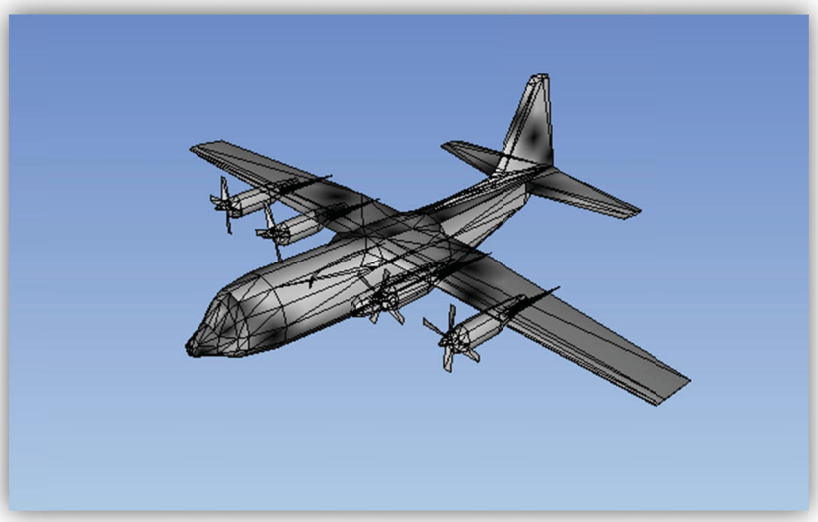

Figure 1: 3D model used in the baseline scenario.

The engagement is the following: a basic threat is pursuing a platform during 3 seconds; the platform is tracked by the missile throughout the simulation. The initial engagement geometry is depicted in Figure 2 while the initial image of the platform by the missile's seeker is shown in Figure 3. Notice that the platform's infrared signature is a generic one, using fake temperatures: the purpose of the simulations conducted during this task was only to test the scene generation frame rate performances.

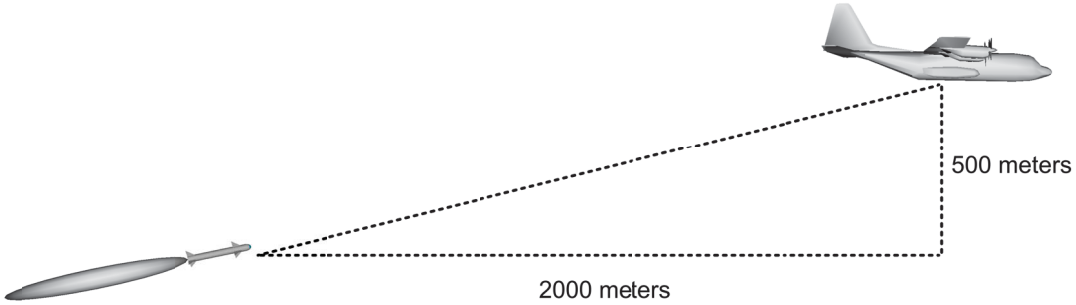

Figure 2: Initial engagement geometry.

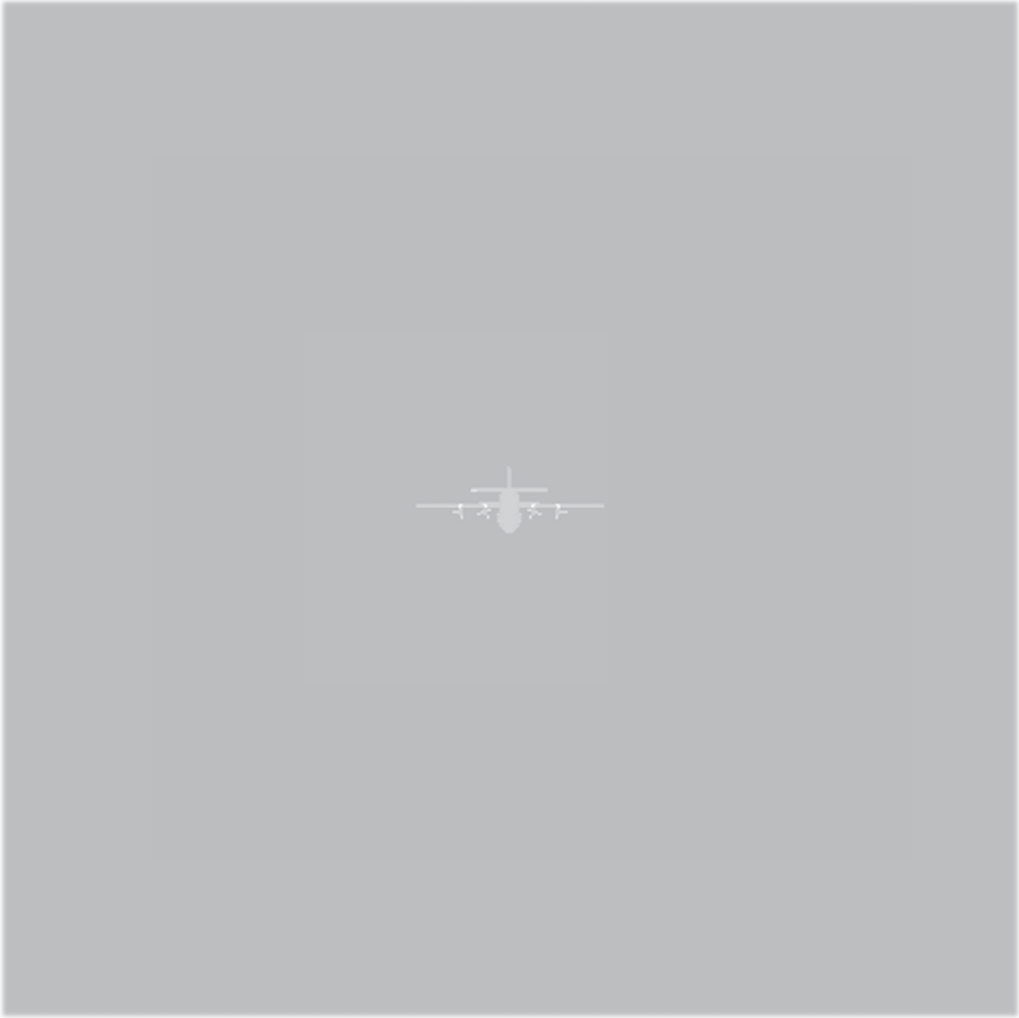

Figure 3: Initial missile's seeker view.

2.3 Performance Profiler

Performance Profiler is a custom timing profiler available within the KARMA simulation framework. It is based on an intrusive technique where the user has to insert code into the method or block of code to be profiled. This approach allows a detailed profiling (smaller granularity) as it is not limited to the existing methods. Performance Profiler uses Windows high-resolution timer (based on *QueryPerformanceCounter* and *QueryPerformanceFrequency*) to determine the duration of a portion of code.

At the beginning of the task, the Performance Validator[1] tool was also tested. The first step was to profile all methods implied in the scene generation process to find those for which the optimization effort will be done. After that, only the chosen methods are profiled each time an optimization is done. When Performance Validator was used to profile every method, the results obtained were not accurate and it was really long to run a scenario. It was then decided

[1] http://www.softwareverify.com/cpp-profiler.php

to use the KARMA tool to identify the longest methods. After this, since the profiling code was already in the KARMA code, the profiling sessions were also realized with the KARMA profiling tool. Switching to Performance Validator could have been done at this point i.e. the results seem to be accurate when profiling a relatively low (~15) number of methods.

2.4 Computing device specifications

The computer used during the performance profiling had the following specifications:

- Intel Core 2 Extreme CPU Q6850 @ 3.00 GHz
- 4.00 GB
- Graphic Card:
    - NVidia 8800 GTX/PCI/SSE2
    - OpenGL 3.3.0
- Windows 7 Professional 64 bits

2.5 KARMA version

The performance profiling and development phases for the optimizations were realized using the KARMA SVN revision 11,336.

3 Profiling and optimizing

3.1 Methods profiled

Table 4 presents the methods that were profiled to find out performance bottlenecks. Obviously, these are not all the methods implied in the scene generation process but those having a non-negligible duration. Notice that Figure 4 shows the hierarchy between these methods.

Table 4: Profiled methods of the scene generation process.

Method	Description
KARMA::Environment:: GetCkBackgroundRadiance	Returns the background radiance (computed by SMART) in the wideband CK format.
KARMA::Environment:: GetCkTransmittance	Returns the atmospheric transmittance (computed by SMART) in the wideband CK format.
KARMA::Environment:: GetCkPathRadiance	Returns the path radiance (computed by SMART) in the wideband CK format.
KARMA::ImagingSensor::Run	Generates an image of the scene (i.e. calls *KARMA::SceneGenerator3D::GetSceneImage*). The imaging sensor represents the imaging device (of the missile) capable of capturing a scene. Additional processing can be added (via other models/parts) to process its output and produce more representative detector data. The *Run* method is called at each period, for every model taking part into the simulation.
KARMA::IRSG::GenerateScene	Generates an image of the scene for a given point of view and using specific settings (spectral response, field of view, etc.).
KARMA::IRSG::ReadFrameBufferObject	Reads back the attached texture of the framebuffer object.
KARMA::IRSG::UpdateColor	Computes in-band components (*LAtmApp*, *LSunRefApp*, *LUpRefApp*, *LDownRefApp*, *LThermApp*, *transparency*, *nFactor*, etc.) using spectrum values gathered previously.

	These values are the inputs of a fragment shader used to determine the radiance of each fragment of a 3D model.
KARMA::SceneGenerator3D:: GetSceneImage	Generates an image of the scene: prepares the scene in the IRSG, initialize the IRSG parameters (sensor, radiometric values, features of the IRSG (skybox, scattering, terrain)) then calls *KARMA::IRSG::GenerateScene*. This is the method that was monitored to determine the frames per second (FPS).
glReadPixels	OpenGL function which reads a block of pixels from the framebuffer. This function is called via the *KARMA::IRSG::ReadFrameBufferObject* to retrieve the generated image.
osgViewer::ViewerBase::renderingTraversals	OpenSceneGraph (OSG) method which generates an image into the framebuffer. This method is called to begin the rendering process.

3.2 Initial overview

Table 5 presents the results of the performance profiling session prior to any optimizations; while Figure 4 depicts an overview of the main methods (from left to right, the visualization shows a method and its child). The height of each block (i.e. method) is set according to the profiling results and represents the relative duration of the block compared to its parent.

Notice that the *Rendering* block (see Figure 4) is not represented in Table 5: its value is a simple deduction, filling the gap between KARMA*::IRSG::UpdateColor* and its parent (*osgViewer::ViewerBase::renderingTraversals*). Every steps of the rendering phase could be profiled by inserting Performance Profiler code into OpenSceneGraph code. However, at this point, profiling KARMA was more important than studying the impact of third party libraries on the scene generation module. Notice that this could be the case in future tasks.

Table 5: Initial profiling results.

Method	Number of calls	Total time (ms)	% Total time	% IRSG	Average per call (ms)	Average per frame (ms)	Longest call	Longest call (ms)	Shortest call	Shortest call (ms)
glReadPixels	301	1328.01	0.60	6.39	4.4120	4.4120	213	21.947	105	3.319
KARMA::Environment:: GetCkBackgroundRadiance	602	2280.70	1.03	10.98	3.7885	7.5771	7	5.134	538	3.105
KARMA::Environment:: GetCkPathRadiance	602	2223.71	1.01	10.71	3.6939	7.3877	65	4.715	538	3.09
KARMA::Environment:: GetCkTransmittance	602	2225.61	1.01	10.72	3.6970	7.3941	66	4.951	594	3.095
KARMA::ImagingSensor::Run	301	22071.76	10.00		73.3281		1	208.234	112	66.401
KARMA::IRSG::GenerateScene	301	13946.22	6.32	67.15	46.3330	46.3330	1	175.531	24	38.945
KARMA::IRSG::ReadFramebufferObject	301	4857.89	2.20	23.39	16.1392	16.1392	213	36.746	105	13.255
KARMA::IRSG::UpdateColor	301	7127.55	3.23	34.32	23.6796	23.6796	283	29.969	68	20.563
KARMA::SceneGenerator3D:: GetSceneImage	301	20767.41	9.41	100.00	68.9947	68.9947	1	203.114	107	62.799
main	1	220620.73	100.00							
osgViewer::ViewerBase:: renderingTraversals	301	9048.39	4.10	43.57	30.0611	30.0611	1	153.66	58	25.165

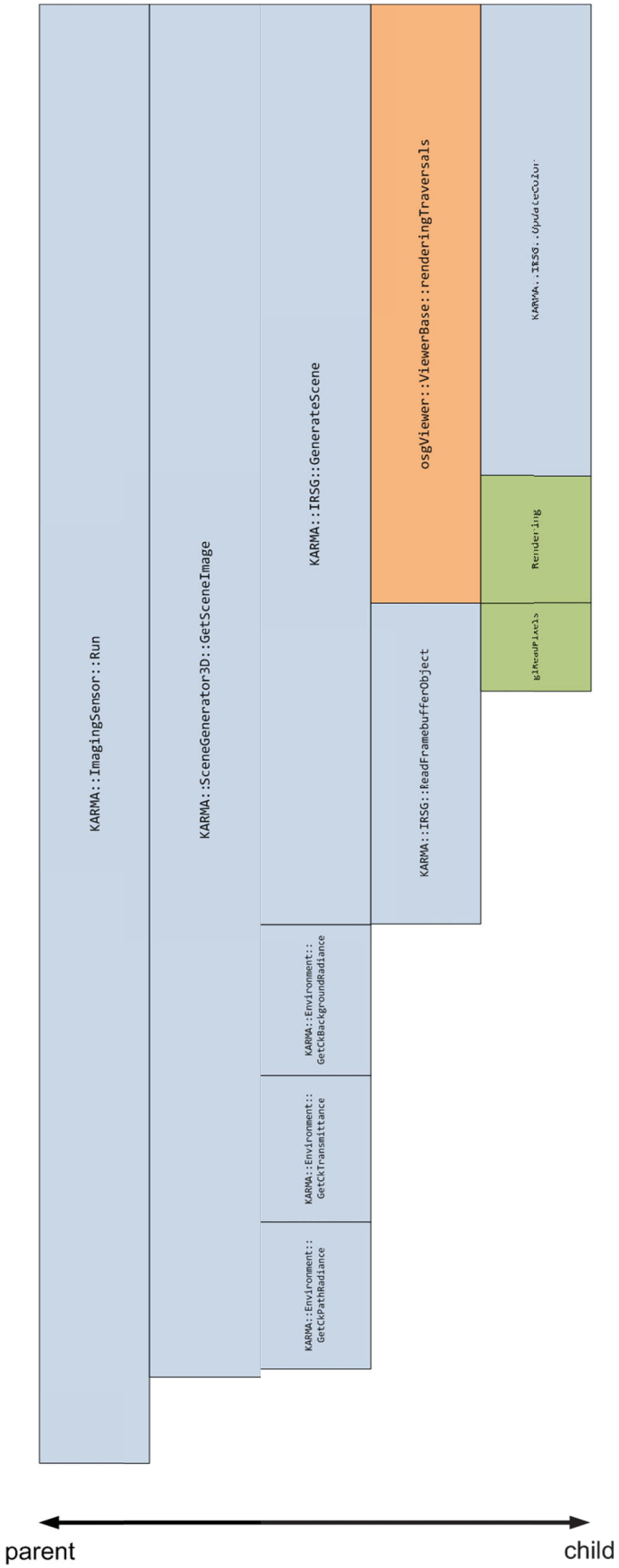

Figure 4: Overview of the relative time for the initial profiling.

3.3 Final results

Table 6 presents the results of the performance profiling session after the development of the optimizations presented in Section 3.4, while Figure 5 depicts an overview of the main methods which were profiled.

The hatched part in *KARMA::ImagingSensor::Run* represents the conversion from the *DataTypes::Image* to a *DataTypes::Matrix*. The latter being the current image format used by the models of the KARMA simulation framework. This conversion is performed to preserve backward compatibility and will not be necessary as soon as the newly created *DataTypes::Image* is used by all models.

The hatched part in *KARMA::IRSG::ReadFrameBufferObject* represents the multiplication of the generated image (radiance) by the *omega* factor to create the image in irradiance. This could eventually be done directly by the various shaders used by the IRSG.

Notice that the first frame (generated at t = 0.00 s.) is longer to generate than the subsequent ones (based on the *Longest Call* column). The time required to produce this image could be removed from the frame rate calculation because it is considered as an “initialization phase”, done prior the simulation begins.

Since the fragment shader could not be profiled, the time spent in this shader to determine the radiance of each fragment of a 3D model has been deduced by removing code and comparing the duration of *osgViewer::ViewerBase::renderingTraversals*. It appears that the shader requires less than 1 ms per frame to execute and the processing is as follows: *LAtmApp* (0%), *LSunRefApp* (35%), *LUpRefApp* (15%), *LDownRefApp* (15%) and *LThermApp* (35%).

For documentation purpose, Appendix A and Appendix B present the results of the performance profiling sessions for the baseline scenario with the antialiasing mechanism (supersampling 4x and ZAA 512) activated.

Table 6: Final profiling results.

Method	Number of calls	Total time (ms)	% Total time	% IRSG	Average per call (ms)	Average per frame (ms)	Longest call	Longest call (ms)	Shortest call	Shortest call (ms)
glReadPixels	301	1075.89	0.55	11.58	3.5744	3.5744	264	15.43	169	3.109
KARMA::Environment:: GetCkBackgroundRadiance	602	2244.22	1.14	24.15	3.7279	7.4559	60	4.812	548	3.094
KARMA::Environment:: GetCkPathRadiance	602	2177.72	1.11	23.43	3.6175	7.2349	24	4.662	578	3.087
KARMA::Environment:: GetCkTransmittance	602	2180.50	1.11	23.46	3.6221	7.2442	59	4.71	568	3.09
KARMA::ImagingSensor::Run	301	10644.77	5.42		35.3647		1	92.713	279	31.885
KARMA::ImagingSensor::Run [convert Image to Matrix]	301	1325.89	0.68		4.4049		12	6.576	46	4.065
KARMA::IRSG::GenerateScene	301	2592.27	1.32	27.89	8.6122	8.6122	1	60.469	35	7.54
KARMA::IRSG::ReadFramebufferObject	301	1208.03	0.62	13.00	4.0134	4.0134	264	16.054	212	3.512
KARMA::IRSG::ReadFramebufferObject [* omega]	301	125.16	0.06	1.35	0.4158	0.4158	251	1.171	165	0.216
KARMA::IRSG::UpdateColor	301	226.19	0.12	2.43	0.7514	0.7514	1	2.181	23	0.652
KARMA::SceneGenerator3D:: GetSceneImage	301	9294.11	4.73	100.00	30.8774	30.8774	1	87.937	272	27.626
main	1	196409.32	100.00							
osgViewer::ViewerBase:: renderingTraversals	301	1319.27	0.67	14.19	4.3829	4.3829	1	54.842	51	3.662

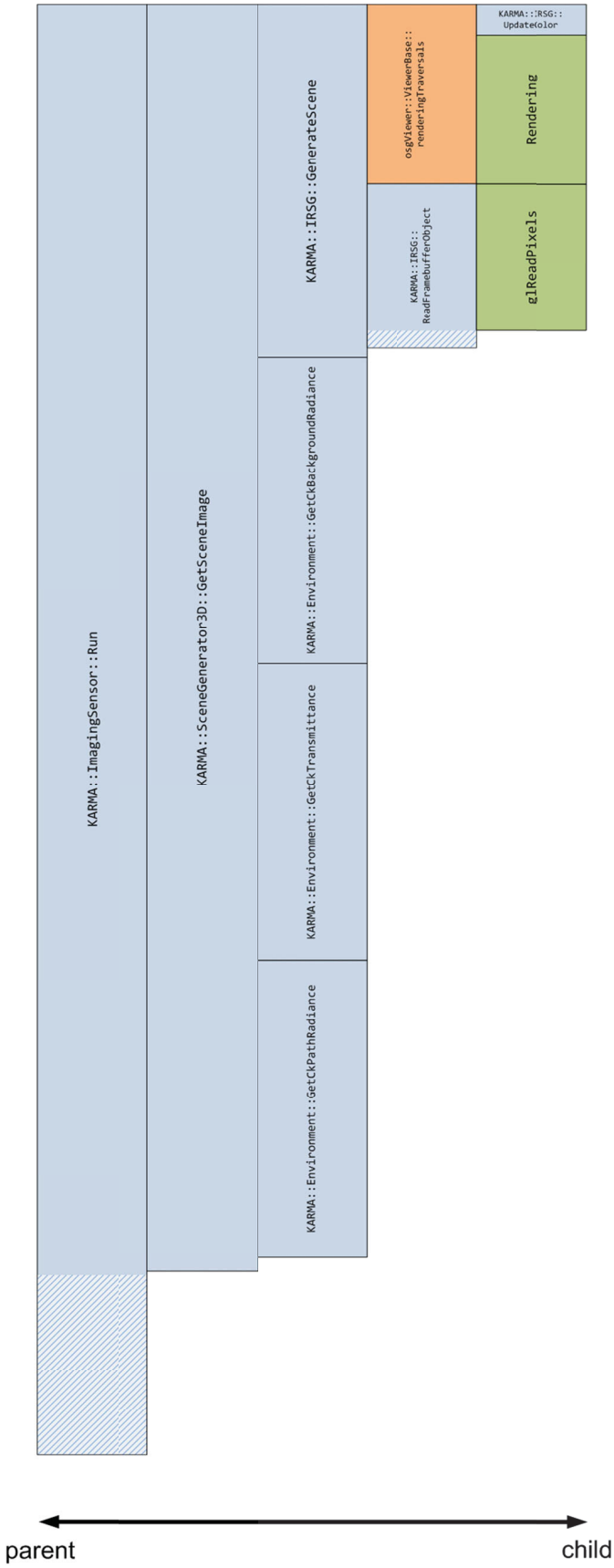

Figure 5: Overview of the relative time for the final profiling.

Table 7 shows a summary of the frame rate, in frames per second (FPS), obtained before and after the optimizations.

Table 7: Summary of the frame rate.

	Frame rate (FPS)
Prior to optimizations	14.5
After optimizations	32.4
After optimizations without 1st frame	32.6

3.4 Improvements

This section describes the improvements to the KARMA framework and IRSG module to improve the scene generation process. Notice that the choice was made to focus on the wideband mode (not the spectral mode) during the optimization phase. The wideband mode is assumed to be the most commonly used in the KARMA simulations.

3.4.1 Radiance computation

The method responsible for the radiance computation has been refactored to relocate some processing and remove useless computations. This method, *IRSG::UpdateColor*, computes in-band components for each facet of a 3D model that could be seen by the camera (i.e. facets that have not been culled) using input spectrums of the IRSG. These components are used by a fragment shader during the rendering process to determine the radiance (or color) of each fragment of a 3D model.

First of all, the validation of temperature (*IRColor*) and material (*IRMaterial*) indices has been relocated at initialization phase. These indices are associated to each facet of a 3D model and are used to compute in-band components: thermal apparent radiance, reflected sun apparent radiance, reflected upward/downward flux apparent radiance, and facet transparency. As soon as a 3D model is added to the IRSG, the *IRSG::GroupVisitor* class is used to validate that these indices are available as user-defined properties for each *osg::Geode* of the 3D model. Additionally, the combinations of the temperature and material indices are computed and stored as a *CombinedIndex* property for further reference in the *IRSG::UpdateColor* method.

In order to reduce the processing load, pre-computations have been extended in the *IRSG::UpdateColor* method. Firstly, as for the thermal apparent radiance, the other in-band components are now computed once and stored in a lookup table (LUT) for each entity. These values are related to the temperature and/or material indices and are retrieved from the LUT when a facet having the same properties has been already processed for a specific entity. This improvement allows reducing computations substantially since entities (or 3D models)

commonly have multiple facets sharing the same properties. Secondly, spectral operations that are independent from the temperature and/or material indices are now pre-computed for each entity. Note that this optimization is limited to the wideband mode since the spectral mode is likely to introduce result disparities if in-band components are present, due to a change in the order of the spectrum operations. Therefore, the following operations are pre-computed (for thermal and solar components of the wideband mode) and used the first time an in-band component is computed: reflected sun apparent radiance, reflected upward/downward flux apparent radiance, and atmosphere apparent radiance.

Finally, the properties used by the fragment shader to compute the radiance are sent sparingly. These properties were set at the lowest level of the scene graph (*osg::Drawable*) using *uniforms*. However, it appears during performance profiling that uniforms are time consuming. As some of these properties are the same for all facets of an entity, uniforms have been relocated. Some uniforms are now set at the entity level (atmosphere apparent radiance, sun azimuth/elevation, sun position and zenith direction) while others are still set at the facet level (thermal apparent radiance, facet transparency, NAngle factor and reflections). Additionally, some *uniforms* have been gathered as *osg::Uniform::FLOAT_VEC3* to limit the uniforms sent to the fragment shader: reflection contributions (sun apparent radiance, upward and downward flux apparent radiance) and facet values (thermal apparent radiance, facet transparency and NAngle factor).

3.4.2 Specialized data type

A new KARMA data type has been created to avoid useless conversion or memory copy. The rendering process creates an image of the scene into the framebuffer object and the OpenGL function *glReadPixels* is used to read these values into an array of *float* values. A specialized object (*DataTypes::Image*) is now used to store an image instead of using a local array and copying (and converting to double) the data in a *DataTypes::Matrix*, element by element. The image (*DataTypes::Image*) is simply initialized to allocate sufficient memory (float values) and this memory is used by the *glReadPixels* function (i.e. there is no memory copied after the execution of *glReadPixels*).

3.4.3 Radiance image on demand

An irradiance image is calculated instead of calculating the radiance and irradiance images each time an image is generated. The models in a KARMA simulation mainly use irradiance images while the radiance image is mainly used for debugging purpose (when images are saved on disk). As it is costly to maintain both radiance and irradiance images (image copy and conversion), the choice was made to generate a radiance image on demand. Thus, the radiance image is only converted from the irradiance image when a model, or a component, requests this format to the IRSG.

3.4.4 Reduced OpenGL calls

The combined effect of removing unnecessary OpenGL calls related to materials (by calling *removeAttribute(osg::StateAttribute::MATERIAL*) on the statesets of *osg::Geode* and *osg::Drawable*) and optimizing the uniforms (see Section 3.4.1) has reduced the number of OpenGL calls from 3,947,025 to 932,845. The statistics related to OpenGL calls before the optimizations are presented in Appendix C; while those generated after the optimization phase are presented in Appendix D.

3.4.5 SMART spectral sensor

The use of SMART (*KARMA::SmartAdapter*) has been revisited to allow using this library at its full potential in wideband mode. Previously, a SMART sensor was always created to receive spectral requests (i.e. the spectral mode) from the *KARMA::Environment*. Thus, in a simulation involving only a sensor using the wideband mode, two SMART sensors (1 wideband and 1 spectral) were created. In such a case, even if only the wideband one was used, having a spectral sensor in the *KARMA::SmartAdapter* increased the duration of the *SMART::Update* method which is called for each generated frame. Therefore, SMART sensors are now created when it is relevant to the current state of the simulation (i.e. the generic SMART spectral sensor will only be created if necessary).

3.4.6 Varia

Some parts of the IRSG that are not presented in this report were also reviewed. A major refactoring of the IRSG is still necessary (to increase encapsulation and modularity) but some improvements were done like:

- remove useless and duplicate code;
- fix memory leaks; and
- remove useless memory copy (e.g. methods inside the IRSG return a pointer on the generated image instead of always recopying the image. In fact, the image shall just be copied when a model wish to modify and process the image).

Before the end of the task, a review of the zoom antialiasing (ZAA) mechanism was started. The shader used by the ZAA to downsample a texture was generalized (for various downsampling factors) and simplified. However, on an ATI graphics card, the shader produced inaccurate results. Given the remaining time for the task, the shader was reverted to the preceding version.

4 Discussion

4.1 Rendering

Some useful tools can be used to profile the rendering part. OpenGL debuggers such as GLIntercept[2] and gDEBugger[3] were used during this task.

The GLIntercept tool is simple to use. A modified version of OpenGL32.dll needs to be copied in the folder where the application to be debugged is located. The user can also set some logging preferences in a configuration file. After this, the software to be profiled just has to be started normally and some logs will be created during its execution. GLIntercept allows logging all OpenGL calls during the execution of a program and gathering statistics such as those depicted in Appendix C and Appendix D.

gDEBugger is a GUI tool that can also be helpful during the debugging phase. It allows having a list of all OpenGL functions called during the execution of a program, set breakpoints, pause the execution, see the content of framebuffers, etc. One particular interesting feature is that gDEBugger can track redundant OpenGL state changes occurring during the simulation. For instance, there are still approximately 25% of the OpenGL calls made during the execution of the base scenario that consist in redundant state changes. Different solutions could be considered to reduce this phenomenon. OSG is responsible of doing the OpenGL calls. As a high level rendering library, OSG is producing redundant state changes in order to be flexible. A study of OSG code may allow identifying parameters that can be set to reduce the number of unnecessary calls. Replacing OSG with direct OpenGL calls shall also be investigated. Figure 6 shows a screenshot of gDEBugger during a debugging session.

[2] http://code.google.com/p/glintercept/
[3] http://www.gremedy.com/

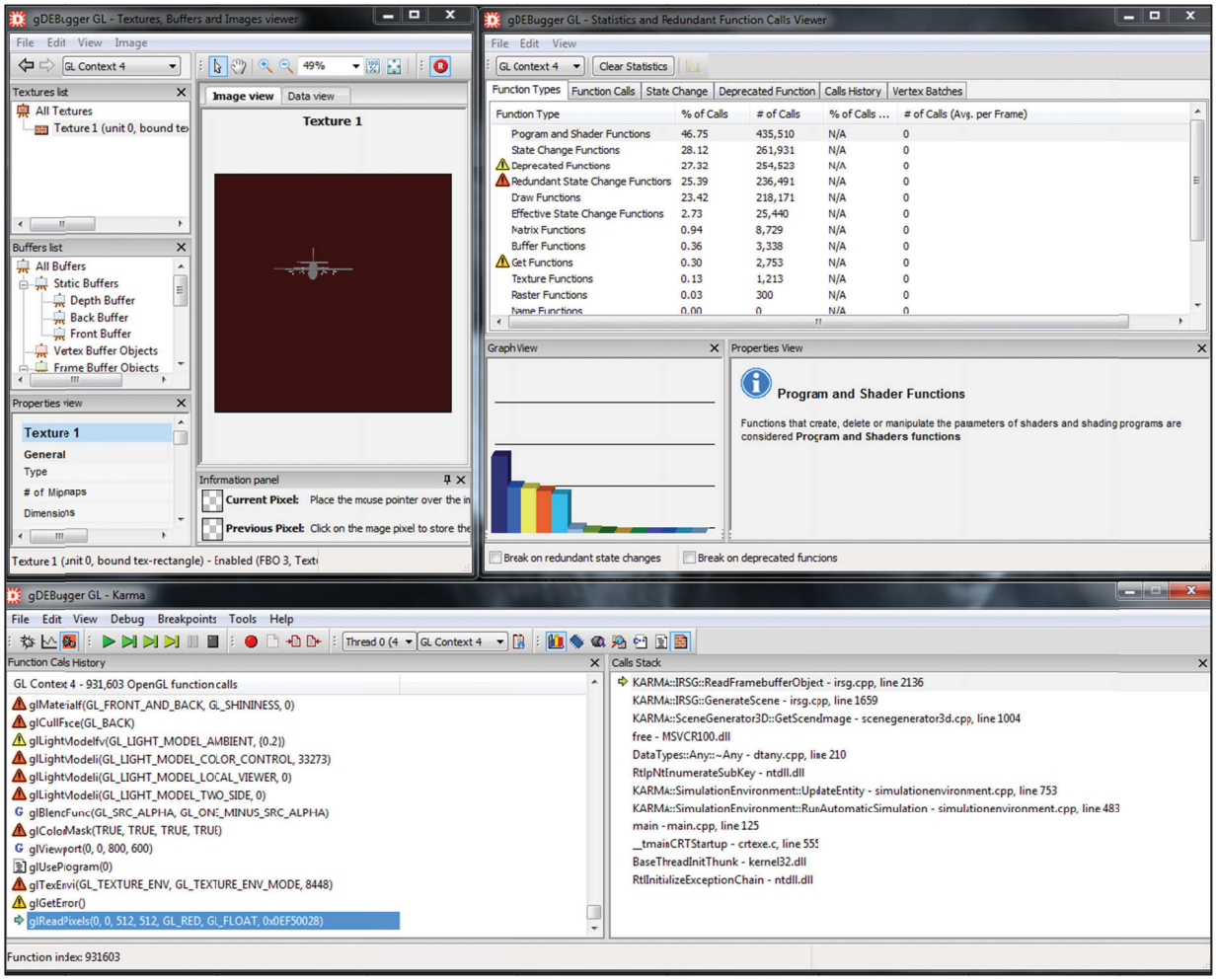

Figure 6: A screenshot of gDEBugger during a profiling session.

4.2 Scene graph structure optimization

During this task, it appears that the structure of the underlying scene graph of the 3D model is not optimal. Each *osg::Drawable*, i.e. facet, of the 3D model is associated to a different *osg::Geode*. Thus, there are as many geodes as drawables. A good optimization would be to regroup facets having the same temperature (*IRColor*) – material (*IRMaterial*) combination under the same geode. It would have 2 main benefits: 1) the scene graph will be smaller and its traversal faster; 2) values computed for a particular temperature – material combination could be sent to the shader at the *Geode* level instead of doing it at the *Drawable* level (see Section 3.4.1). A tool could be developed to optimize automatically the scene graph or it could be done at initialization when a 3D model is added to the IRSG.

4.3 Level of detail

The more *IRColor* and *IRMaterial* combinations a model has, the more complex computations are required to deduce the apparent radiance of each fragment of an entity (see Section 3.4.1). Therefore, the level of detail of the signature (i.e. database) could be reduced, by cutting down the total number of combinations, to gain extra frames per second. Similarly, the level of detail of the 3D models could be reduced by cutting down the number of polygons since the IRSG processes every facet.

Finally, the size of the image could also be reduced: the time required to read the framebuffer and process an image is directly related to the size of the image. A quick test showed that reducing the image from 512x512 pixels to 401x401 pixels, using the baseline scenario presented in Section 2.2, increases the performance of the IRSG by 5%. Obviously, this is a tradeoff between precision and execution speed.

4.4 Parallelism

An important upgrade would consist in using the central processing unit (CPU) at its maximal capacity. At this moment, the IRSG is very sequential: every step of the rendering process is conducted one after the other. The performance of the IRSG would increase if some of these computations were done in parallel, in separate threads.

Some tests were conducted using the Parallel Patterns Library[4] (PPL) that is included in Visual Studio 2010 (development environment). This library simplifies the implementation of parallel processing and offers three algorithms:

- *parallel_for*;
- *parallel_for_each*; and
- *parallel_invoke*.

Firstly, the *parallel_for* algorithm has been tested for the *DataTypes::Image* to *DataTypes::Matrix* conversion (inside *Image::ToMatrix* method) for the baseline scenario. This conversion occurs each time a frame is generated (i.e. 301 times). Table 8 presents the average duration for two image sizes. In both cases, the use of the *parallel_for* algorithm speeds up the conversion process by more than a factor 2.

Table 8: Image conversion without vs. with the parallel_for algorithm.

Image size (pixels)	Average duration (ms)	
	Without *parallel_for*	With *parallel_for*
512x512	5.56	2.21
1024x1024	23.89	10.03

[4] http://msdn.microsoft.com/en-us/library/dd492418.aspx

Notice that the performances of such a parallelism are tightly related to the implementation. Indeed, there is an overhead in the management of the threads, so the load processing of each thread must be substantial in order to achieve a significant performance gain.

Secondly, the *parallel_invoke* algorithm has been tested to parallelize the use of SMART (inside *KARMA::SceneGenerator3D::GetSceneImage* method) since the SMART library is supposed to be multi-thread ready. The computation of the path radiance, sun irradiance and upward/downward fluxes has been parallelized while preserving results integrity. Once again, the performances were compared using the baseline scenario. The use of *parallel_invoke* allowed to speed up the computation of these values by almost a factor 2, from 5.60 ms to 3.42 ms. These calls to SMART have been parallelized easily but the code must be refactored in order to parallelize other atmospheric components at the entity level (i.e. atmospheric transmission and scattering). As the computation of the atmospheric components for each entity represents nearly the half duration of a frame rendering, parallelization could increase the frame rate by 25%.

The use of the PPL library will help reaching higher frame rates. The major bottlenecks during rendering process could also benefit from parallelism.

4.5 Replace SMART

For time critical execution, SMART shall be replaced as it is too long to get radiometric values from it. A mechanism based on pre-defined LUT could be developed where values are gathered in these tables and a method responsible for the interpolation between these values is defined.

Another reason to use LUT is connected to the fact that a scenario can contain multiple entities. In this case, the portion of the scene generation process related to acquiring values from SMART will be multiplied compared to what is depicted in Table 6 and Figure 5.

4.6 Asynchronous read pixels / Double buffering

Reading pixels from the framebuffer (*glReadPixels*) takes almost as much time as doing the rendering (see Figure 5). It is possible to read pixels asynchronously and avoid blocking the CPU while waiting for direct memory access (DMA) transfer on *glReadPixels* calls. One of the easiest way to achieve this is to take advantage of using multiple pixel buffer objects (PBO) to asynchronously download pixels from the framebuffer into a mapped PBO while the CPU process pixels from an earlier PBO. This is an important optimization that requires to consider memory vs. performance tradeoff since PBO needs a lot of memory.

There are multiple ways to implements PBOs, but if the IRSG is refactored to use parallelism (see 4.4), it could be perfectly timed with the rendering pipeline and thus, avoid rendering while downloading pixels (which cause a locking problem called OpenGL pipeline stall).

4.7 Dynamic memory allocation

This optimization targets the KARMA simulation framework globally but could also have important impact on the IRSG. Dynamic memory allocation (heap) can create fragmentation and is much slower than stack memory allocation (heap allocation calls are usually forwarded up to the operating system and depending of the platform, it can be 100x slower).

Using specialized memory allocator (ex: Nedmalloc[5], ptmalloc[6] and Hoard[7]), it is possible to prevent fragmentation and improve memory allocation to get huge performance improvement (especially when using multiple threads (see 4.4)).

Finally, it is possible to get comparable performance to stack allocation out of heap allocation by using memory pool to allocate a large memory block at the beginning of the application. It can be somewhat complicated to implement without causing fragmentation and other problems, but several open source implementations exists (such as Boost memory pool[8]).

4.8 Graphics processing unit

Exploiting the graphics processing units (GPU) for additional operations shall also be investigated. As an example, the supersampling mechanism could use the GPU to avoid costly memory transfers from the GPU to the CPU. Indeed, it is possible to downsample directly via the GPU instead of downsampling via a custom method which accesses elements of the array to find the appropriate samples. Also, reading back the image in the CPU, via the *glReadPixels* function, would be faster since the size of the image is already at the final dimension (i.e. downsampled).

One of the improvements done (see Section 3.4.3) consists in only keeping the irradiance image and converting it in radiance on demand. If it comes to the point where the two formats are always necessary, the shaders already in place in the IRSG could manage the conversion. For example, the radiance of a fragment could be placed in the red channel while the results of the radiance multiplied by the conversion factor (radiance to irradiance) could be placed in the blue channel.

[5] http://www.nedprod.com/programs/portable/nedmalloc/
[6] http://www.malloc.de/en/
[7] http://www.hoard.org/
[8] http://www.boost.org/doc/libs/release/libs/pool/

Finally, instead of transferring the image from the GPU to the CPU, models processing this image could be modified to process data directly in the GPU. However, this approach would tighten dependencies between models and the IRSG.

5 Conclusion

During this task, the IRSG was profiled to grasp a better understanding of the underlying mechanisms as well as where the time is spent during the scene generation process. According to the available time for the whole task AT-10, many optimizations were done. The improvements were tested using a baseline scenario, and the results indicate that the frame rate has been increased from 14.5 FPS to 32.4 FPS, with SMART still in the loop. Replacing SMART with various LUT should allow obtaining 100 FPS in a similar scenario as the one used during the performance profiling session.

There are still many possible optimizations, like those discussed in this document, which would increase the frame rate of the IRSG. The rendering part shall be investigated in order to try reducing the number of OpenGL calls. An OpenGL debugger tool used with the profiled scenario indicated that 25% of the rendering calls are related to redundant OpenGL state changes. As discussed previously, some solutions could be considered to reduce the effect of this expensive mechanism. Additionally, it would be interesting to conduct performance profiling sessions using scenarios involving multiple targets (platforms and flares), antialiasing mechanisms and complex backgrounds (skybox and terrain). This would allow determining different bottlenecks and possible optimizations for standard simulations.

References

[1] Ross, V. and Dion, D., "SMART and SMARTI: visible and IR atmospheric radiative transfer libraries optimized for wide-band applications", Proc. SPIE 8014, 80140S (2011).

Appendix A

Profiling results (after optimization phase) with supersampling 4x activated.

Table 9: Profiling the base scenario with supersampling 4x (after the optimizations).

Method	Number of calls	Total time (ms)	%Total time	% IRSG	Average per call (ms)	Average per frame (ms)	Longest call	Longest call (ms)	Shortest call	Shortest call (ms)
glReadPixels	301	18816.35	7.66	33.92	62.5128	62.5128	144	87.605	68	57.024
KARMA::Environment:: GetCkBackgroundRadiance	602	2247.12	0.91	4.05	3.7328	7.4655	35	5.657	550	3.092
KARMA::Environment:: GetCkPathRadiance	602	2179.48	0.89	3.93	3.6204	7.2408	8	4.85	532	3.086
KARMA::Environment:: GetCkTransmittance	602	2182.88	0.89	3.93	3.6260	7.2521	35	5.076	594	3.083
KARMA::ImagingSensor::Run	301	57055.93	23.23		189.5546		1	245.041	253	180.231
KARMA::ImagingSensor::Run [convert Image to Matrix]	301	1550.50	0.63		5.1512		153	10.335	220	4.565
KARMA::IRSG::GenerateScene	301	48772.81	19.85	87.91	162.0359	162.0359	1	211.109	253	155.556
KARMA::IRSG::ReadFramebufferObject	301	47093.91	19.17	84.89	156.4582	156.4582	208	185.845	253	150.352
KARMA::IRSG::ReadFramebufferObject [downSample]	301	28145.56	11.46	50.73	93.5068	93.5068	208	98.326	196	92.609
KARMA::IRSG::ReadFramebufferObject [* omega]	301	118.64	0.05	0.21	0.3942	0.3942	166	0.916	134	0.318
KARMA::IRSG::UpdateColor	301	282.66	0.12	0.51	0.9391	0.9391	1	3.296	264	0.868
KARMA::SceneGenerator3D:: GetSceneImage	301	55479.19	22.58	100.00	184.3162	184.3162	1	239.512	253	175.3
main	1	245665.29	100.00							
osgViewer::ViewerBase:: renderingTraversals	301	1611.47	0.66	2.90	5.3537	5.3537	1	49.2	38	4.895

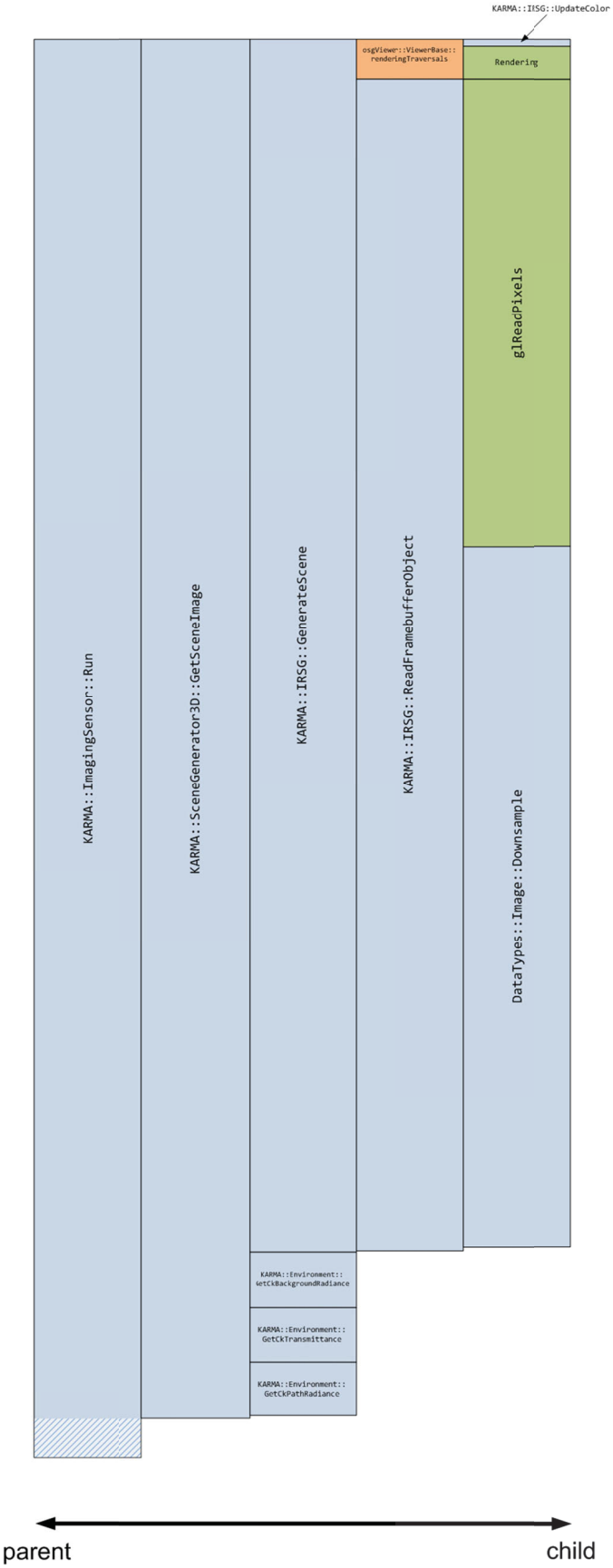

Figure 7: Relative times (with the optimizations) when the supersampling 4x is activated.

This page intentionally left blank.

Appendix B

Profiling results (after optimization phase) with ZAA 512 activated.

Table 10: Profiling the base scenario with zoom antialiasing (ZAA) 512 (after the optimizations).

Method	Number of calls	Total time (ms)	%Total time	% IRSG	Average per call (ms)	Average per frame (ms)	Longest call	Longest call (ms)	Shortest call	Shortest call (ms)
glReadPixels	301	1809.72	0.90	14.04	6.0123	6.0123	122	21.516	18	4.931
KARMA::Environment:: GetCkBackgroundRadiance	602	2267.16	1.12	17.59	3.7661	7.5321	161	5.159	536	3.1
KARMA::Environment:: GetCkPathRadiance	602	2199.96	1.09	17.07	3.6544	7.3088	168	5.053	537	3.093
KARMA::Environment:: GetCkTransmittance	602	2201.67	1.09	17.08	3.6573	7.3145	3	4.902	554	3.093
KARMA::ImagingSensor::Run	301	14355.69	7.11		47.6933		1	125.698	162	42.575
KARMA::ImagingSensor::Run [convert Image to Matrix]	301	1439.46	0.71		4.7823		83	10.4	274	4.084
KARMA::IRSG::GenerateScene	301	6122.49	3.03	47.50	20.3405	20.3405	1	91.838	37	16.411
KARMA::IRSG::ReadFramebufferObject	301	1950.65	0.97	15.13	6.4806	6.4806	122	21.978	37	5.338
KARMA::IRSG::ReadFramebufferObject [* omega]	301	133.42	0.07	1.04	0.4433	0.4433	139	0.865	279	0.215
KARMA::IRSG::UpdateColor	301	292.59	0.15	2.27	0.9721	0.9721	1	3.461	203	0.863
KARMA::SceneGenerator3D::GetSceneImage	301	12890.60	6.39	100.00	42.8259	42.8259	1	120.527	162	38.244
main	1	201776.44	100.00							
osgViewer::ViewerBase::renderingTraversals	301	4004.04	1.98	31.06	13.3025	13.3025	1	84.462	45	10.247

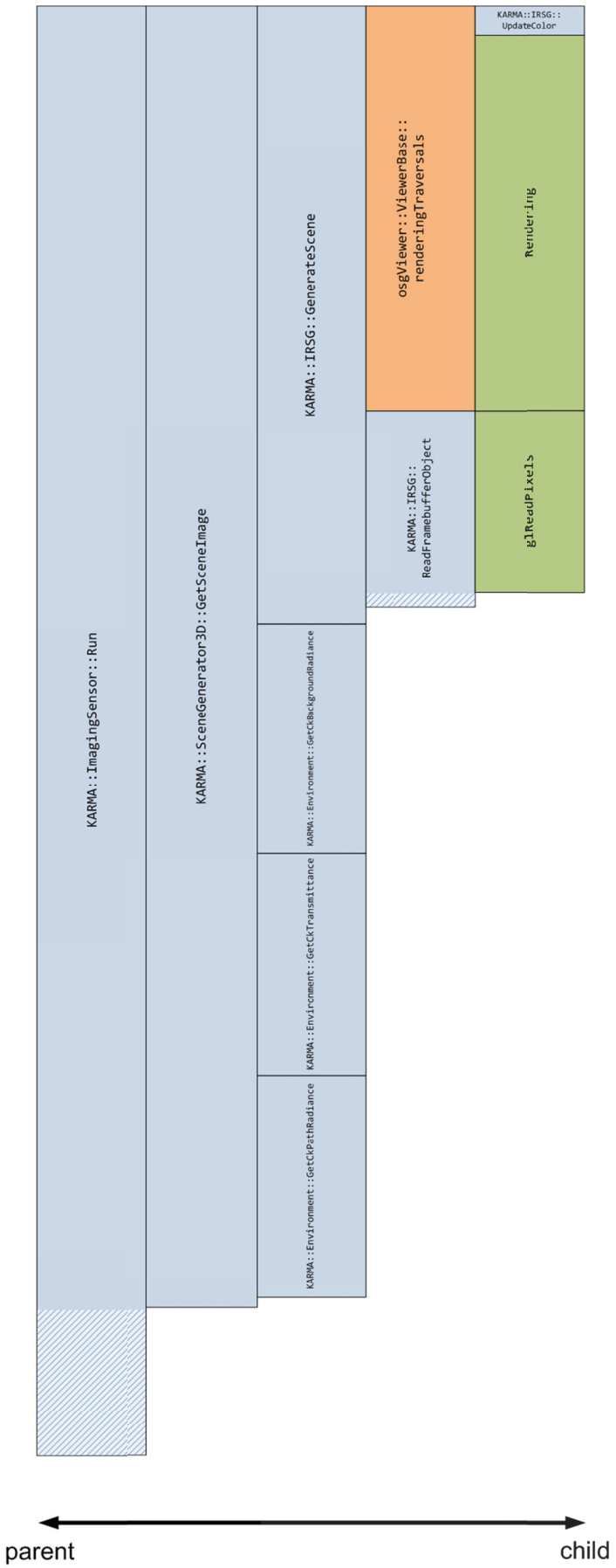

Figure 8: Relative times (with the optimizations) when the ZAA 512 is activated.

This page intentionally left blank.

Appendix C

OpenGL function calls before the optimization phase.

```
======= OpenGL function call statistics ==========
 Total GL calls: 3947025

======= OpenGL function calls by call count ==========
glUniform1fv ................................ 1508241
glMaterialfv ................................ 864862
glUniform3fv ................................ 430926
glDisable ................................... 222176
glColor4fv .................................. 217873
glMaterialf ................................. 216366
glCallList .................................. 215764
glCullFace .................................. 208330
glEnable .................................... 6412
glLightf .................................... 4515
glLightfv ................................... 4515
glMatrixMode ................................ 3913
glUseProgram ................................ 3615
glLoadIdentity .............................. 3612
glLightModeli ............................... 3612
glBindFramebufferEXT ........................ 2110
glColorMask ................................. 2107
glGetError .................................. 1809
glViewport .................................. 1806
glNewList ................................... 1507
glEndList ................................... 1507
glVertexPointer ............................. 1507
glDeleteLists ............................... 1507
glGenLists .................................. 1507
glBlendFunc ................................. 1505
glDrawBuffers ............................... 1207
glNormalPointer ............................. 1206
glDrawArrays ................................ 1206
glAlphaFunc ................................. 1204
glLoadMatrixd ............................... 1204
glLightModelfv .............................. 1204
glTexEnvi ................................... 1204
glClear ..................................... 903
```

Appendix D

OpenGL function calls after the optimization phase.

```
======= OpenGL function call statistics ==========
 Total GL calls: 932845

======= OpenGL function calls by call count ==========
glUniform3fv ................................ 431522
glCallList .................................. 215761
glCullFace .................................. 208198
glDisable ................................... 6584
glEnable .................................... 6584
glLightfv ................................... 4515
glLightf .................................... 4515
glMatrixMode ................................ 3913
glUseProgram ................................ 3615
glLightModeli ............................... 3612
glMaterialfv ................................ 3612
glLoadIdentity .............................. 3612
glColor4fv .................................. 2711
glBindFramebufferEXT ........................ 2110
glColorMask ................................. 2107
glGetError .................................. 1809
glViewport .................................. 1806
glNewList ................................... 1507
glEndList ................................... 1507
glVertexPointer ............................. 1507
glDeleteLists ............................... 1507
glGenLists .................................. 1507
glBlendFunc ................................. 1505
glDrawBuffers ............................... 1207
glNormalPointer ............................. 1206
glDrawArrays ................................ 1206
glMaterialf ................................. 1204
glAlphaFunc ................................. 1204
glLoadMatrixd ............................... 1204
glLightModelfv .............................. 1204
glTexEnvi ................................... 1204
glClear ..................................... 903
glClearDepth ................................ 903
```

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.) LTI software & engineering 825 Boul. Lebourgneuf, Bureau 204 Québec, Canada G2J 0B9	2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.) UNCLASSIFIED (NON-CONTROLLED GOODS) DMC A REVIEW: GCEC JUNE 2010

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

Air Defence System: Profiling and Optimizing the KARMA IRSG Module

4. AUTHORS (last name, followed by initials – ranks, titles, etc. not to be used)

Labrie, M.-A., Rouleau, E., Desmeules, M.

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)	6b. NO. OF REFS (Total cited in document.)
July 2012	50	1

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Contract Report

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – Valcartier
2459 Pie-XI Blvd North
Quebec (Quebec)
G3J 1X5 Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.) Project 13nb	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.) W7701-083373-AT10
10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.) LTI-ADS-2012-1	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.) DRDC Valcartier CR 2012-070

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

Unlimited

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))

Unlimited

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

The main objectives of task AT-10, as part of contract W7701-083373, was to profile the performance of the infrared scene generator (IRSG) module of the KARMA simulation framework, pinpoint bottlenecks, optimize critical portions of code through basic corrective measures, and update the SMART (Suite for Multi-resolution Atmospheric Radiative Transmission) library to the latest version. The work has been carried out from January 2012 to March 2012. The profiling has been limited to a basic scenario using the IRSG and the wideband mode. The code optimizations allowed to double the frame rate for this scenario. Some optimization avenues were investigated and could be realized as a part of another task or contract.

Les objectifs de la tache AT-10, du contrat W7701-083373, étaient d'effectuer le profilage des performances du module de génération de scène infrarouge (IRSG) de l'environnement de simulation KARMA, d'identifier les éléments critiques, d'optimiser certaines portions critiques par des mesures correctives de base, et de mettre a jour la librairie de calculs atmosphérique SMART (Suite for Multi-resolution Atmospheric Radiative Transmission). Les travaux ont été effectués de janvier 2012 à mars 2012. Le profilage des performances a été limité à un scénario simple utilisant l'IRSG en mode large-bande. Les optimisations au code ont permis de doubler la cadence pour un tel scénario. Des avenues ont aussi été identifiées pour une éventuelle deuxième phase d'optimisation.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

IR Scene Generation; Code Optimisation; Graphics Processing Units

Defence R&D Canada

Canada's Leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
De science et de technologie pour
la défense et la sécurité nationale

DEFENCE DÉFENSE

www.drdc-rddc.gc.ca