Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

Master`s Thesis

Visualization of human's body internal tissues using shaders in simulation medicine

Nikolai Spiridonov

Supervisor: Ing. Karel Frajták, Ph.D.

Study Program: Open Informatics
Field of Study: Software Engineering
May 24, 2018

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Spiridonov**	Jméno: **Nikolai**	Osobní číslo: **474705**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačů**		
Studijní program:	**Otevřená informatika**		
Studijní obor:	**Softwarové inženýrství**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Visualization of human's body internal tissues using shaders in simulation medicine

Název diplomové práce anglicky:

Visualization of human's body internal tissues using shaders in simulation medicine

Pokyny pro vypracování:

This diploma thesis deals with task of visualization of human`s inner tissues and organs in medical simulators. The main purpose of the work is to develop algorithms that allow to unload the CPU and shift the load from it to the graphics card.
Diploma thesis is divided into 2 parts:
1) visualization of liquid tissues
2) visualization of soft bodies
First part includes visualization of blood in laparoscopic simulator and flow of contrast agent inside the vessels in endovascular simulator. Second parts deals with the visualization of coagulated tissues, fat and membrane tissues, common visualization of organs with blood and coagulation in simulator of laparoscopic surgery.
This work is not only about rendering, it is also about some "logical" problems arising in the development of medical simulators, which are most often solved with the help of CPU. Visualization in simulation medicine is a very computationally complex task, which, using the approaches proposed in the work, can be solved very effectively

Seznam doporučené literatury:

1) William E. Lorensen, Harvey E. Cline , // CG vol.21, no.4, July 1987
2) Bernardo P. Carneiro, Arie E. Kaufman , // SIGGRAPH'96
3) Vaclav Skala , Conference on Scientific Computing 2000, // pp. 368 - 378.http://www.emis.de/journals/AMUC/_contributed/algo2000/skala.pdf
4) Yiyi Wei, Stéphane Cotin, Jérémie Allard, Le Fang, Chunhong Pan, et al.. Interactive Blood-Coil Simulation in Real-time during Aneurysm Embolization. Computers and Graphics, Elsevier, 2011, Visual Computing in Biology and Medicine, 35 (2), pp.422-430. ?10.1016/j.cag.2011.01.010?. ?hal-00688443?
5) Guillaume Kazmitcheff, Hadrien Courtecuisse, Yann Nguyen, Mathieu Miroir, Alexis Bozorg-Grayeli, et al.. Haptic Rendering on Deformable Anatomical Tissues with Strong Heterogeneities. Eurohaptics 2014, Jun 2014, Versailles, France. Springer, 2014. ?hal-01025614?
6) François Dervaux, Igor Peterlik, Jérémie Dequidt, Stéphane Cotin, Christian Duriez. Haptic Rendering of Interacting Dynamic Deformable Objects Simulated in Real-Time at Different Frequencies. IROS - IEEE/RSJ International Conference on Intelligent Robots and Systems, Nov 2013, Tokyo, Japan. IEEE, 2013. ?hal-00842866?
7) Hoeryong Jung, Stéphane Cotin, Christian Duriez, Jérémie Allard, Doo Yong Lee. High Fidelity Haptic Rendering for Deformable Objects Undergoing Topology Changes. EuroHaptics - Haptics: Generating and Perceiving Tangible Sensations International Conference, Jul 2010, Amsterdam, Netherlands. Springer, 6191, pp.262-268, 2010, LNCS. ?10.1007/978-3-642-14064-8_38?. ?hal-00688913?

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Karel Frajták, Ph.D., katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **16.04.2018** Termín odevzdání diplomové práce: **25.05.2018**

Platnost zadání diplomové práce: **30.09.2019**

Ing. Karel Frajták, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání Podpis studenta

Acknowledgements

I would like to express my sincere gratitude to all of my teachers, especially E.A. Turilova, E.M. Karchevskiy and N.R. Boukharaev for their patient and guidance in my study. I want to specially thank my family: E.N. Spiridonova and V.V. Timoshenko for their supporting and encouraging me in a way that no other ever could.

Abstract

Modern real-time 3D applications are very complex and sophisticated software. Especially difficult is the development of software for medical simulators. Often developers try to improve the performance of their software trying to infinitely optimize their programs. However, sooner or later there comes a point when optimization does not give significant advantages. In such cases, you should look for other resources, which can increase system perfomance. The idea proposed in this paper suggests that it is possible to effectively use the graphic device even for some logical calculations if one approaches the development of special algorithms correctly. Usually programmers do not spend their time studying shader technologies and general computing on a graphical device, however this master`s thesis shows that this approach has an advantage. It allows us to use previously free resources.

Keywords: simulation medicine, HLSL, shaders, GPGPU, real-time, 3d applications.

Contents

List of figures

Introduction

The modern medicine cannot be imagined without the latest technologies. They actively are implemented into practice of health care, at the same time, obviously, use of innovations carries to improving of quality of the provided services. Application of the latest development allows to reduce significantly risks and losses when carrying out difficult interferences. However, the reverse of the medal is complication of procedures that certainly causes even more requirements to the finite performer — human. One of the most striking examples is the technology of laparoscopic interferences. The laparoscopy is a type of surgery in which the operational interference occurs through small holes (trocars) in a body of the humanю A camera and the necessary tools entered to cavity through those holes. This method of conducting operations is advantageously characterized by low traumatism and a short recovery period. Small trocars are tightened much faster than large longitudinal incisions of cavitary surgery, with no postoperative scars. However, a laparoscopic intervention is much more difficult in execution than a classical operation. The absence of a direct autopsy presents much higher demands on the surgeon. The expert has to be guided ideally in space, perfectly known with the anatomy of human`s body. In addition, the very movement of the cutting surfaces inside the patient occurs in a key that is absolutely not intuitive for the surgeon - the movement of the instruments is inverted and it passes in a limited area. All these features require the surgeon to have additional skills that are difficult to learn. Often, training is also hampered by ethical norms and laws. Training on real examples is usually impossible. In such cases, special medical simulators are used that help the cadet to gain the necessary knowledge, develop necessary skills and psychological stability.

In versatile real-time applications, modules are divided into two types: logic and renderer. Logic modules are the mathematical center of an application. The calculation of the physical model, statistics and algorithms for performing the actions is located here. The rendering modules are engaged in visualizing the data received in the logic modules. It is considered that in the logical part of the application all main and most important calculations are made. At the stage of visualization, programmers just solves tasks of drawing the objects. However, after analyzing the program, we note that the application is in constant "time trouble". Higher the frame rate per second the better the 3d application. Always and in everything the programmers are looking for new ways to accelerate computation. Such applications are always load CPU`s cores on a maximum. Most often, to increase efficiency of the program developers looking for methods of optimization of the algorithm itself more and more. However, it is worthwhile to understand that constant optimization becomes more and more complicated and hard. The more time you spend on improving the algorithm, the smaller the result it brings. It is impossible to constantly accelerate algorithms, there is always a limit, which is incredibly difficult to achieve. To get the improvement, you need to find other resources, a free space that would give a qualitative leap in improving the program's performance. If you analyze modern applications (even 3d applications), then the CPU is always loaded to the maximum. The video card is often left free, dealing with not too complicated rendering tasks. This is due to the fact that most programmers simply do not know how to write programs for the GPU. And the very development of such algorithms requires a somewhat different approach to solving problems, it is necessary to take into account the architecture of the device, its strengths.

The problems solved in this work came during the development of two simulators: laparoscopy and endovascular surgery. These tasks can be divided into 2 groups: visualization of liquid tissues and visualization of soft tissues of the human body. At first glance, it would seem that the visualization problem is the most important part of the medical simulator software, but this is not so. In medical simulators, the most resources are allocated to the logical part of the application, directly on a miscalculation of physical model and after that not too much resources left on visualization tasks. Of course, the first thing that a potential customer sees on an advertising brochure or at the first meet to a simulator is a picture, and it certainly must be of high quality, but the final decision is made on the basis of logical and physical correctness of the work. Therefore, the requirement for the algorithms given below is the minimum operating time with satisfactory visualization quality. On average, during the rendering time of each task within a frame, no more than 3 ms is allocated, which obviously does not allow using, for example in the visualization of liquids, the most advanced techniques developed within leading graphic libraries. Another limitation is caused by the fact that the tasks being solved are the so-called part of the general logic of the system, that is, they are used in all exercises and modules, since they provide the basic work on the simulation of human vital activity (blood flow, soft tissue visualization and etc.). These modules are present in all models of simulators, both old and new. And if new simulators use the most modern equipment as computational parts, then older versions often have very old hardware installed. The development policy also assumes updating on very old computers, as a consequence, it is impossible to use some of the latest technologies (for example, using shaders above the third level, geometric shaders, etc.). All these requirements make the task very difficult at its objectively high social significance - as the quality of training of surgeons ultimately determines the life of our loved ones.

Given the above facts, the purpose of our work is to develop algorithms for visualizing liquid (blood, contrast) and soft (coagulated and conventional) tissues using shader technology to shift the load from the CPU to the graphics card. This diploma thesis consists of the introduction, theoretical, algorithmic, practical chapters, conclusions. At the end of the work are the applications in which you can find additional images, source code for shaders and some functions on c ++. For high-level logic used proprietary software that requires too expensive license. Logic implemented on this level

isn’t the point of our work, but their behavior is represented as schemas in the relevant part.

Chapter 1

Theoretical aspects of technologies and field of applications of algorithms

1.1 General information about the scope of algorithms

1.1.1 Specificity and main challenges of modern simulation medicine

There are different approaches in the design of modern medical simulators:

1) Modelational. With this approach, the simulator itself provides the model of the object under study. This could be a box with elastic bands in the form of an object under study or a similar device physically modeling the body.

2) Simulational. In this case, virtual reality is simulated, which recreates the object under study in space. And skills and abilities are transferred to the student by using various virtual reality technologies (monitors, virtual reality glasses, etc.)

The advantage of the first approach is in the simplicity of manufacture. Simulation of the behavior of instruments and models is based on real laws of physics, which makes it as realistic as possible to convey this to the student. However, with such an approach it is practically impossible to reconstruct a sufficiently plausible scene of a real operation, and for the realization of even two different anatomies, in fact, two different simulators are needed. The development of the second type of simulators requires much greater expenditure. It is necessary to implement a whole complex of hardware and software. Special complex sensors for determining the position of instruments in space, the most complex software that simulates the studied environment. However, with this approach it is possible to achieve substantially better learning outcomes for the future surgeon. The special software responsible for the simulation allows to train skills on real patient models (rehearsal of operations) and can also give young cadets advice on how to conduct an intervention, tips and directions in choosing a tool, etc.

Algorithms proposed in the relevant part of this work are used in the second type of simulators. Software is the heart of such a system. It must solve a whole range of complicated tasks: tracking the logic of operations, performing an intelligent system of hints, simulating and visualizing physical processes, and so on. All these operations are obviously extremely difficult to develop and resource-intensive, require large computing power. In addition, most processes run in real time, the software must produce a high rate of frames per second - it affects the quality of the simulation and, ultimately, the quality of the training of the future surgeon. The priority is the plausibility of the simulation; the very visualization of processes should provide an acceptable level of schematics. From the above, it follows that the CPU is loaded to the maximum of possibilities and one of the few ways to reduce the load is to shift the computations from the CPU to the graphics card. This approach is the basis of this diploma thesis and all the algorithms somehow rely on this idea. However, the development of such algorithms requires somewhat different approaches to solving problems than classical algorithmization. The graphical card imposes significant restrictions on the format of the data to be returned, the algorithms used and the many familiar functions that are spent almost without cost in the case of programming for the CPU.

1.1.2. Simulator of endovascular interventions.

Endovascular surgery is a type of surgical intervention performed on the blood vessels by percutaneous access. This intervention is carried out under the supervision of methods of radial imaging using special instruments [8]. The main advantage and peculiarity of the method is that all interventions occur through a small incision-puncture on the human body. Then, under the control of special X-ray equipment, the surgeon inserts operating instruments into the circulatory system of the person, allowing to conduct the procedure (catheters, conductors, stentgraphs, special cylinders, etc.) During the execution of operation, only local anesthesia is performed, and the recovery period after operations, if it was performed without complications, is from 1 to 3 days.

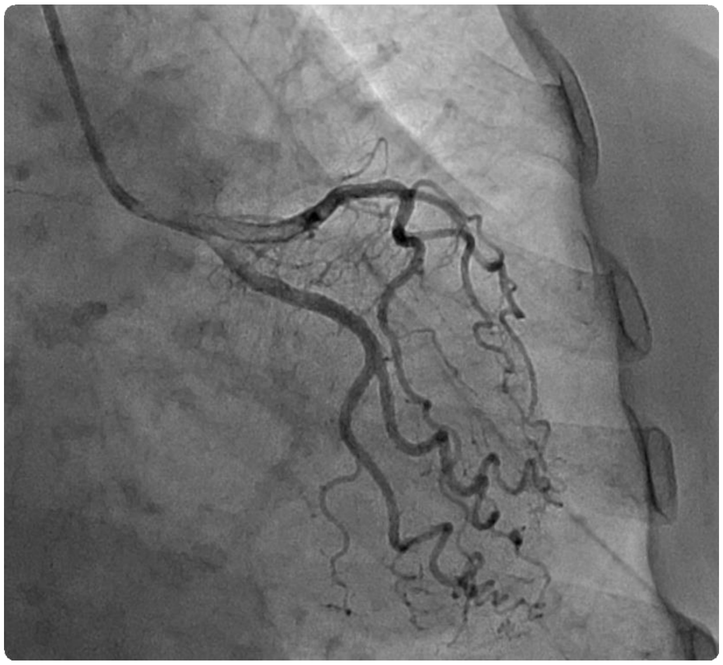


Figure 1. X-ray image during endovascular intervention.

The surgeon inserts tools into the human body and monitors the process using an X-ray, the tools are made of special materials, allowing them to be seen on an X-ray. However, the blood and blood vessels are not visible on the x-ray, so the doctor through the catheter introduces the patient a special radiopaque fluid, which, mixing with the blood, allows the surgeon to navigate.

The task that we will consider within the functionality of this simulator is visualization of the movement of contrast fluid (we will call it just contrast below) in blood vessels. Contrasting liquid mixing with blood moves along the vessels and it looks like water movement through pipes, this effect is the basis of the simulation.

1.1.3 Simulator of laparoscopic interventions

Laparoscopy is a method of modern surgery, in which the operation is performed through small holes — trocars. Through them, inside the patient is entered camera and additional tools with which all manipulations are carried out. This method of operation advantageously has low recovery time of the patient, however, the requirements for surgeon-laparoscopist significantly higher. The doctor observes the image transmitted from the camera and should be perfectly guided in space inside of human`s body. The specifics of the development of the laparoscopy simulator is very complex and involves large and non-trivial visualization tasks. The cadet should observe the organs themselves, while they can be stained with blood, subjected to coagulation. The bodies themselves inside the cavity differ in their structure, can provide both dense and opaque structures, and thin and barely noticeable fat membranes. So, we can highlight the main problems that have been solved in this work in the framework of laparoscopy simulator:

- Blood visualization
- Visualization of the coagulated tissue
- General visualization of soft tissues of internal human`s body organs

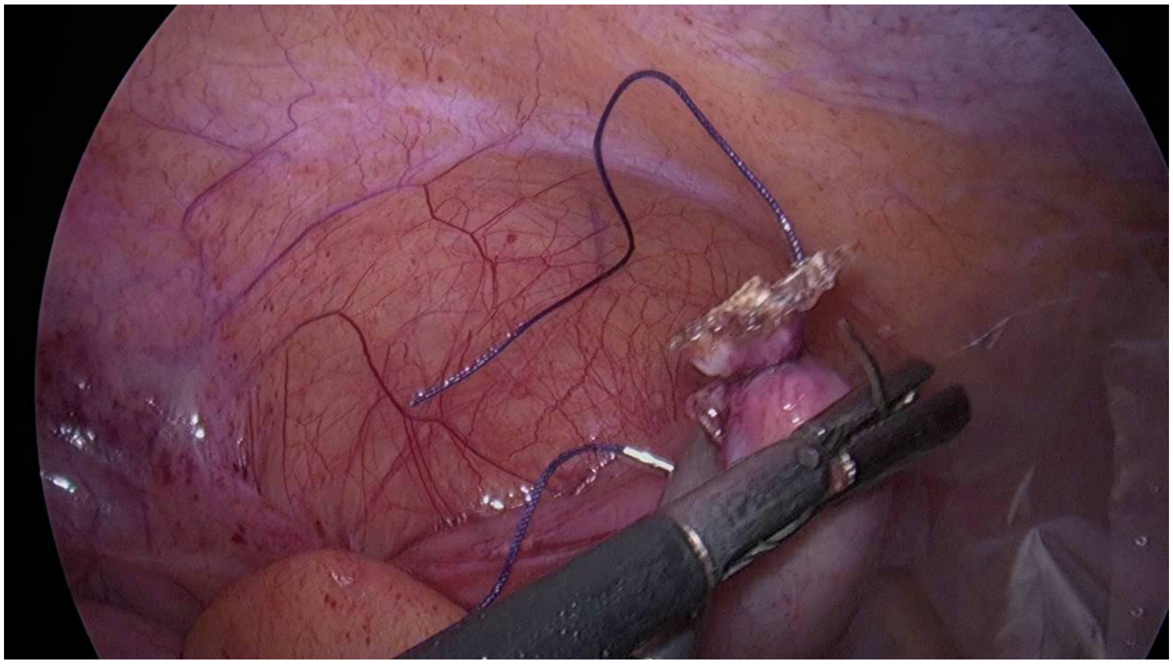


Figure 2. Image from laparoscope camera

The essence of the tasks is in principle clear from the names, their specificity and concretization will be given in the next Chapter, but **it's necessary to** give some comments and explanation about what coagulation is. Coagulation - physical and chemical process of adhesion of small particles of dispersed systems to larger ones under the influence of adhesion forces. This effect underlies the ability of blood to curtailed. In the process of carrying out a laparoscopic operation, large bleeding can be discovered. To avoid this, the surgeon coagulates the tissues with help of special electrical tools. The tissue under such influence changes its structure and color, and it should be visualized on the screen.

1.2 Visualization of **human’**s body internal tissues

1.2.1 Visualization of liquid tissues

Visualization of various liquids is a well-studied problem of computer graphics. The most common approach to solving this problem is algorithms based on the idea of a particle system [1] – [6]. Their essence lies in the fact that the movement of the liquid is simulated by primitive particles — most often spheres. Sources of such particles (fluid emitter) are created, which generate new particles. Each such molecule is influenced by the laws of physics and all of them together form the surface of water. These algorithms provide excellent visual results, especially in terms of highlights and light refraction, however, they require a lot of computing resources. In our case, light rays do not pass through the liquid at all, and the picture itself does not involve any difficulties in terms of tracing the rays. Also, the simulation itself of a large number of particles, involves an advanced collision detection and raycasting, in which there is no need. In view of these reasons, another approach based on previously calculated data was invented. The essence of it is that to calculate the three-dimensional geometrical object of the future fluid to be visualized. We will use a three-dimensional object of the blood vessels themselves, stained with contrast, and some additional information. The mathematical model in this approach is based on the construction of a graph for the initial data and the calculation of fluid`s flow paths. This approach greatly simplifies the calculations and does not require complicated collision calculations. In fact, we have the original three-dimensional object of arteries and the physical engine just determines what part of it is colored at the current frame. The rest of the calculations are transferred to the video card, allowing the load to be shifted from the CPU. The question of computing such an object is trivial and will not be considered in detail, we will assume that based on the model we can get all the information necessary for visualization.

The second problem in this part is the task of visualizing blood from a laparoscopic simulator. As already mentioned above, it can also be solved using the classical approach with a particle system. However, we will apply a more specific method of similar class. According to the logic of its behavior, blood can be conditionally divided into surface and jet. The first type (most often venous phase) is in the body under low pressure and slowly flows down the walls accumulating in cavities. The second type (most often arterial phase) is under great pressure and when opening this type of wound it creates a large jet, which then, in collision with the organs, actually turns into the first type of blood. This task decomposition actually greatly simplifies fluid simulation, and it well reflects the real essence of things. Also, with this approach, we can efficiently use the calculations on video cards to improve performance. The algorithm for jet blood is described in paper [7], in this diploma thesis we will consider the algorithm for visualizing surface blood. As already described above - surface blood is a fluid with a high viscosity. It flows slowly, coloring the organ. This important property tells us that it can be represented in the form of spots on the texture of the organ. The texture of a three-dimensional object is an image that is superimposed on the object in the process of the graphics pipeline and determines its color on the screen. The liquid-type approach allows us to decompose the task, each part itself does not cause any special difficulties and suits very well for the architecture of the graphical devices, as they are designed to work with images and textures.

As we can see, the classical problem of liquid visualization can be presented in a slightly different form. Under special requirements and conditions, we change the task in such a way as to simplify the calculations to the CPU as much as possible. Replacing complex mathematical models with much simpler ones, we unload the CPU, while the main load is shifted to the video card. By slightly changing the conditions of the problem, we get more favorable conditions for implementing the algorithm on the video card, which is ultimately our goal.

1.2.2 Visualization of soft tissues

Physics and visualization of human`s internal soft tissues are the most sophisticated and important part of the laparoscopic simulator. The physical engine is responsible for the plausible simulation of deformations, cuts and many different effects. The task itself is very complex. Also, to the requirements are added that all calculations are performed in real time, so most mathematical models use some approximations, the flaws of which must be covered by the visual part. Within the framework of the soft tissue visualization problem, we will consider 2 problems: visualization and logic of coagulation effects and general visualization of soft bodies.

As already described above, with the help of coagulation surgeons are protected from unwanted bleeding. The surgeon coagulates the desired area, during this process there is a so-called coagulation spot and the body in this area changes its color and physical properties. From the level of coagulation in this location depends whether the bleeding will open or not, but the surgeon does not need to exceed the level of coagulation, otherwise the consequences will not be reversible. Coagulation is most often achieved by exposure of electricity. There are monopolar and bipolar types of coagulation. With monopolar exposure, the spot diverges at a certain radius from the contact point, depending on the power and duration of the exposure. With bipolar coagulation, the electric current passes strictly between the electrodes and affects only the space between them. Realization of the logic of this process is our task. With a classical approach on the central processor, such a problem is solved by creating a structure that stores coagulation information for a three-dimensional object. Each vertex is associated with a certain number - the level of coagulation at a given vertex. Disadvantages of this approach are obvious, three-dimensional objects are very large and can contain a large number of vertices. The storage of such a structure, its processing occupies a significant amount of processor time and memory, this method cannot be called effective. The approach proposed in this diploma thesis is based on the generation of a special logical texture that will contain information about the level of coagulation. Generation of this texture is not difficult, it will be superimposed on the object and provide all the necessary functions for storing data. In this case, it should be noted that the size of the occupied memory does not depend on the number of vertices in the object (for a fixed resolution of the texture) and the generation of such information naturally suits to the architecture of the graphical card.

Another problem under consideration is the final visualization of soft bodies - this is the logical continuation and completion of the effects of blood and coagulation. It is also solved with the help of shader technologies and allows to complete the process of visualization of human soft tissues.

1.3 Specificity of shader programs and brief to graphical pipeline

To write effective Shader programs for graphics cards, you need to clearly understand the architecture and specifics. Video accelerator is a coprocessor designed to unload the CPU. Video card as device should quickly and effectively perform the tasks of the graphics pipeline. From a programmer's point of view, a graphics pipeline is a set of processing steps to produce an image. It is conditionally divided into 2 parts: processing of geometry and processing of fragments (pixels). At the first stage, the coordinates are transformed. To each vertex, the world, view and projection matrix is applied, the calculation of illumination, the generation of texture coordinates, etc. After performing these operations, the primitive layout occurs. The vertices are grouped into triangles and move to the rasterizer. At this stage, the triangle is divided into pixels for which the corresponding values are interpolated (texture coordinates, colors, etc.). After this, texture mapping, alpha test, depth test, mixing and logical operations occur. After processing all methods, the resulting fragment (pixel) is placed in a framebuffer, which is then displayed on the screen.

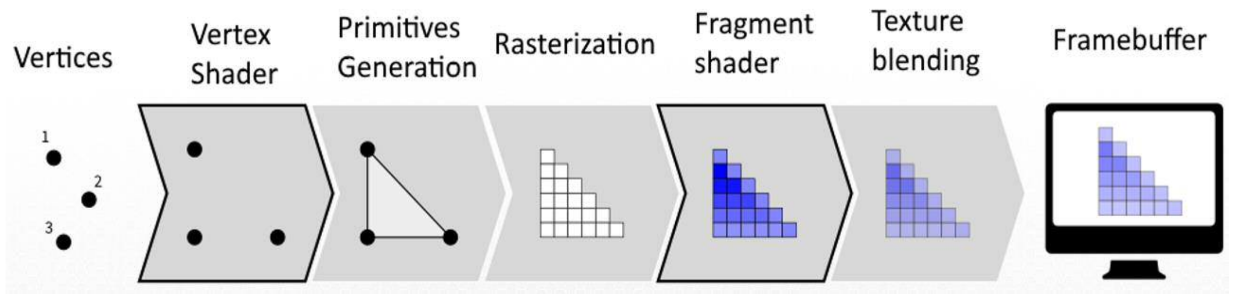


Figure 3. Schematic representation of the graphics pipeline

The graphical pipeline is implemented mainly on the hardware level for obtaining high performance, however, in some of its parts you can insert your own programs, which are called shaders. There are two main types of shaders: vertex and pixel (in OpenGL notation it called fragment) shaders. Note that in the most modern systems there are geometric shaders and special tessellators, but the possibility of using these technologies within the framework used in this work is impossible, because we should use shaders no higher than the third level to support older devices.

The vertex shader is applied to each vertex (a vertex is understood as some structure consisting of some vertex attributes defining a three-dimensional object, such as: vertex position, texture coordinate, color, etc.). During the execution of the vertex shader, you can change and calculate the attributes (normal, position, color, etc.) that will go to the next stage of the graphics pipeline. The pixel shader is aimed to processing image fragments. At this stage, the user can fully control the process of imposing textures, calculating the depth buffer, the color of the pixel. A pixel shader is performed for all pixels from the triangle rasterization phase. A fragment or pixel is a point with window coordinates obtained by the rasterizer. This is the resultant point in the frame buffer, the combination of these pixels ultimately forms the future image [9]. In our work we will rely on the Microsoft shader technologies included in DirectX and specifically the HLSL programming language. The language is similar to C, More information about the HLSL language and graphical pipeline could be found in [9] - [11].

Chapter 2

Algorithms of visualization of internal human tissues

2.1 Visualization algorithms of liquid tissues

2.1.1 Algorithm for visualization of contrast fluid movement

As described above, a contrast fluid is administered to the patient so that the surgeon can distinguish the contours of the vessels with an X-ray. The spread of contrast through the vessels is due to pressure in the circulatory system of man. For X-rays, the blood is transparent, so the contrast movement looks like the usual spreading of liquid in a pipe under pressure. In our algorithm, we have moved away from the classical approach with particle systems. Instead, we will paint the original object of the vessels. Note that this approach significantly simplifies the problem and reduces the requirements for the physical engine that provides the logic of behavior. The logical part receives the input of a three-dimensional object of vessels. Calculates the fluid flow graph over it and passes a part of this object to the visualization module, it paints it according to the density of the contrast at a given point.

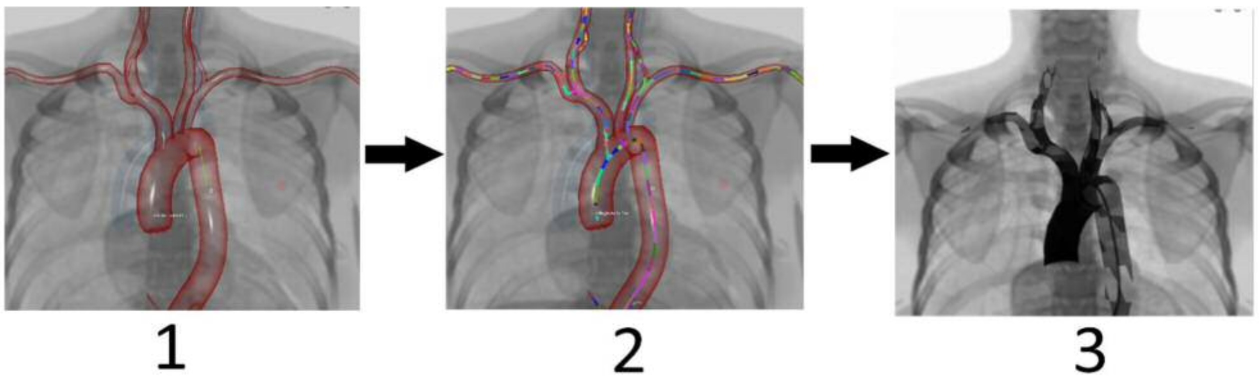


Figure 4. Stages of the logical contrast module

Let's describe in more detail the stages of the logical contrast module:

1) The initial model of arteries is a closed classical three-dimensional object, consisting of vertices distributed along triangles. The model can be drawn by a 3d modeler or obtained from a CT scan of a real patient. [appendix 1.a]
2) The graph constructed by the logical contrast module based on the source model. Arrows indicate the direction of the fluid. [appendix 1.b] The nodes of the graph are the midpoints of the vessel, the "pipe" itself is described around these nodes. The direction of movement is calculated in the direction of decreasing pressure, i.e. from the heart in the arterial part and towards the heart in the venous. From the visualization point of view there is no difference between the arterial and venous phases.
3) The colored object returned by the logical module. In itself, the topology of an object and its geometry do not change with time. Movement is achieved by coloring the vertices in a special way. In the image you can see one frame of such a simulation [appendix 1.b]. At the same time, we put attention to the fact that the object itself is not highly polygonal. In

the eye rushing clear boundaries and sharp transitions, in reality the contrast on the x-ray looks different. The purpose of our algorithm is to the correct visualization of this object. Bringing the image to a view similar to the real picture.

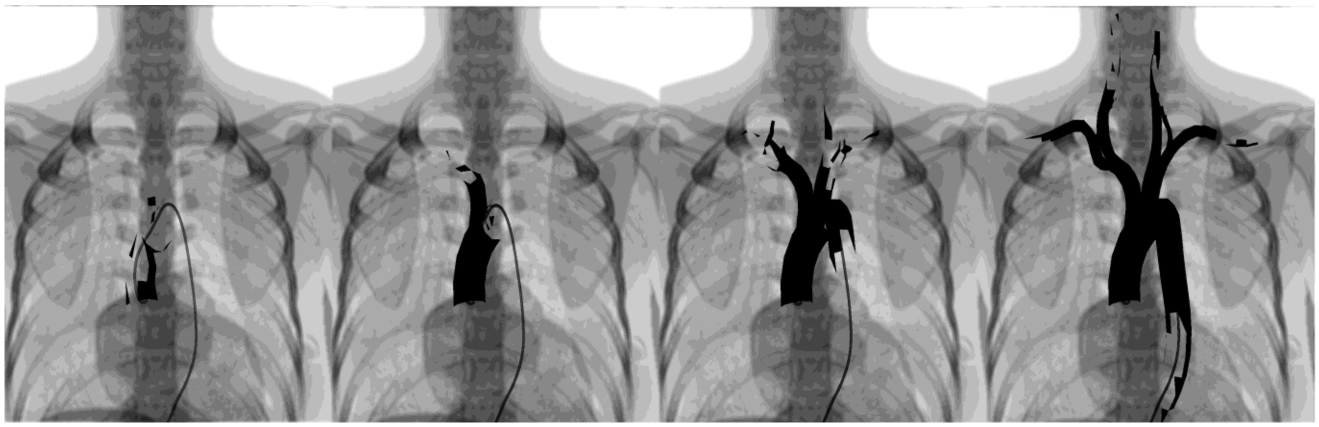


Figure 5. Contrast flow frames

The implementation of such behavior is not a difficult task, which can be calculated in advance. We will not go deeper into the details of the logical part of the contrast, since this is not our goal, but we note that simplification of the logical part necessarily leads to complication of the visualization part, but this is the basis of the idea of this approach, because we need to shift the load to video card.

The three main problems that our algorithm must solve are:

1) Overlapping of brighter areas, which are geometrically covered by less bright
2) Blurring of sharp contrast boundaries
3) The implementation of reflux and extravasate effect

The first problem is the appearance of the so-called overlapping artifact (Figure 6). In the process of drawing an object to the buffer, and later on the screen get those pixels that were last drawn. Often this way of forming an image does not reflect the reality,

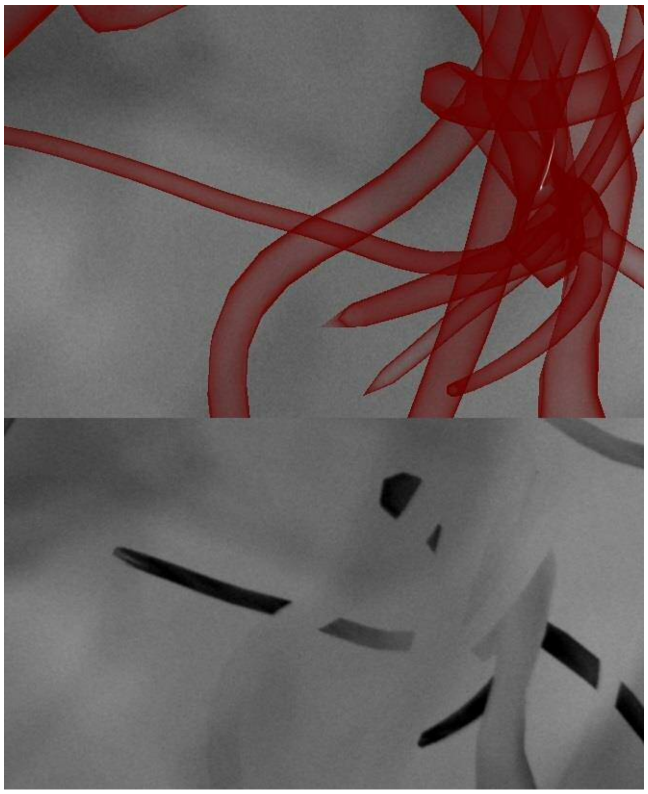


Figure 6. Overlap artifact

because the pixels are generally processed in parallel. In reality, we see those parts of the object that are closer to us and do not overlap with other parts in front of the camera. To ensure this effect, there is a depth test in the graphic pipeline. To ensure its operation, a special depth buffer is created, in a sense an image where a depth value is stored for each pixel. The depth value is a function of the distance from the processed pixel to the camera (most often the inverse distance with normalization). The inverse function is used to increase the accuracy when the object approaches the camera. The depth test is that the pixel is written to the final buffer for rendering if the depth value for this pixel is less than the value already written in the depth buffer. Directly testing the depth on the video card is implemented on a hardware level - this allows device to perform computations extremely effectively but does not give the user desired freedom. In fact, the user can only choose the comparison function (smaller, greater, etc.) using which during the test it will be decided to write a pixel or not [12]. The overlapping artifact arises when less bright areas of the object are geometrically closer to the camera than the brighter ones and overlap those places where the contrast density on the given frame is greater. There are two ways to solve this problem. The first, and most obvious, is a forced write to the depth buffer. We can write our own calculated values in the buffer. This can be some function of the pixel brightness. Implementation of this approach is not difficult and its logic is very simple. However, in this case there are pitfalls. Forcing the depth buffer in some graphics systems can significantly slow down the processing time of the image. You also need resources to allocate your own depth buffer for the contrast object. As a consequence, contrast cannot be drawn in one scene with all other objects, since the buffers of their depth do not match. The contrast object should be rendered in a separate texture with its own depth buffer on the screen at the last stage of the render. In addition to this, it is not always possible to force writing down your values in the depth buffer. Some data may be stored there, which are necessary in the future, and indeed the very possibility of recording in some systems is missing. In this case, another approach, known as the "artist algorithm" [13] [14], can be used. The essence of the algorithm is that first the renderers are delivered to the renderer, which actually lie farther from the camera and gradually the polygons appear closer and closer. In our case, we will first draw the least bright polygons gradually moving to the brightest. In fact, this is a method of topological sorting of pixels based on their brightness values. Note that the approach has a significant obvious disadvantage - the algorithm cannot be executed in parallel for different layers. Therefore, it is recommended to use it only when there is no possibility of direct writing to the user's buffer of depth.

The second problem is the problem of sharp boundaries of contrast. The initial model consists of a finite number of vertices and triangles. The model in some sense is discrete. On the image of the object received at the input, well sensible transitions between triangles with different luminosity appear in the eyes. In reality, the situation is obviously different, there is no clear boundary for the transition of brightness. Note that to solve this problem, a simple blur in the post-processing stage does not work. If we blur the image in this way, we inevitably get an artifact of going beyond the real object.

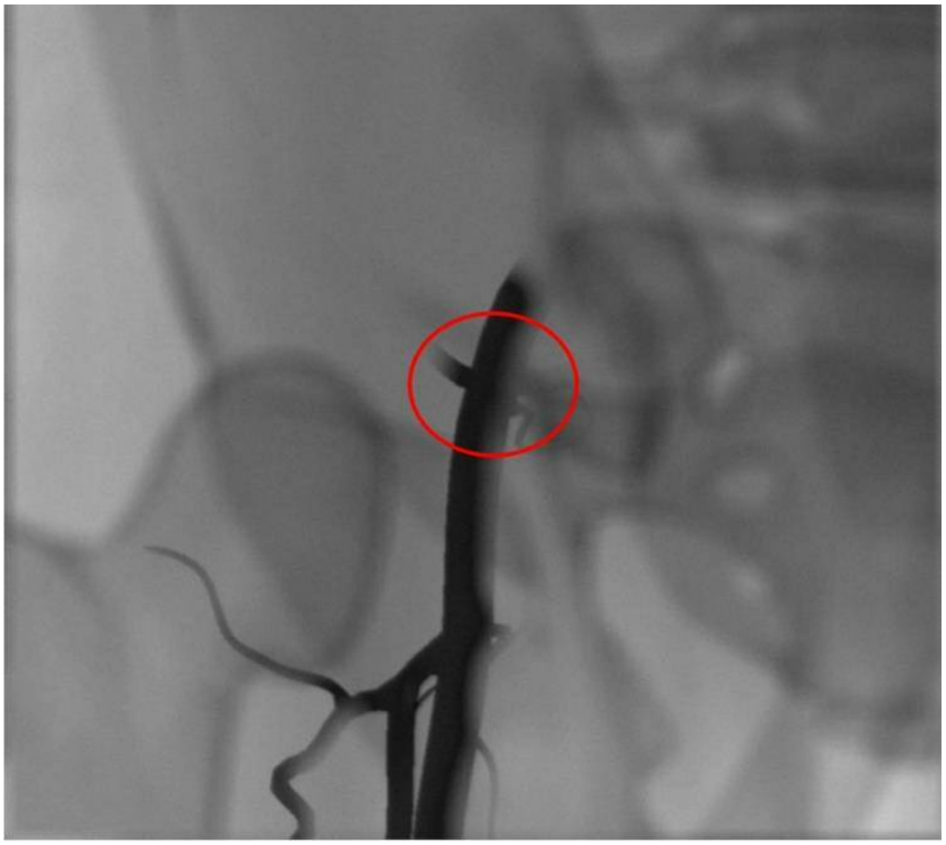


Figure 7. Blurring artifact

This artifact is highly critical and absolutely unacceptable. From figure 7 we can conclude that in this place in the vessel there is a branching, because it seems that there is a contrast. However, in reality this was due to the blurring. Trimming the mask in a particular place failed, because of the overlay of another branch. At the stage of post-processing it is impossible to take into account this fact, since we work only with the obtained image.

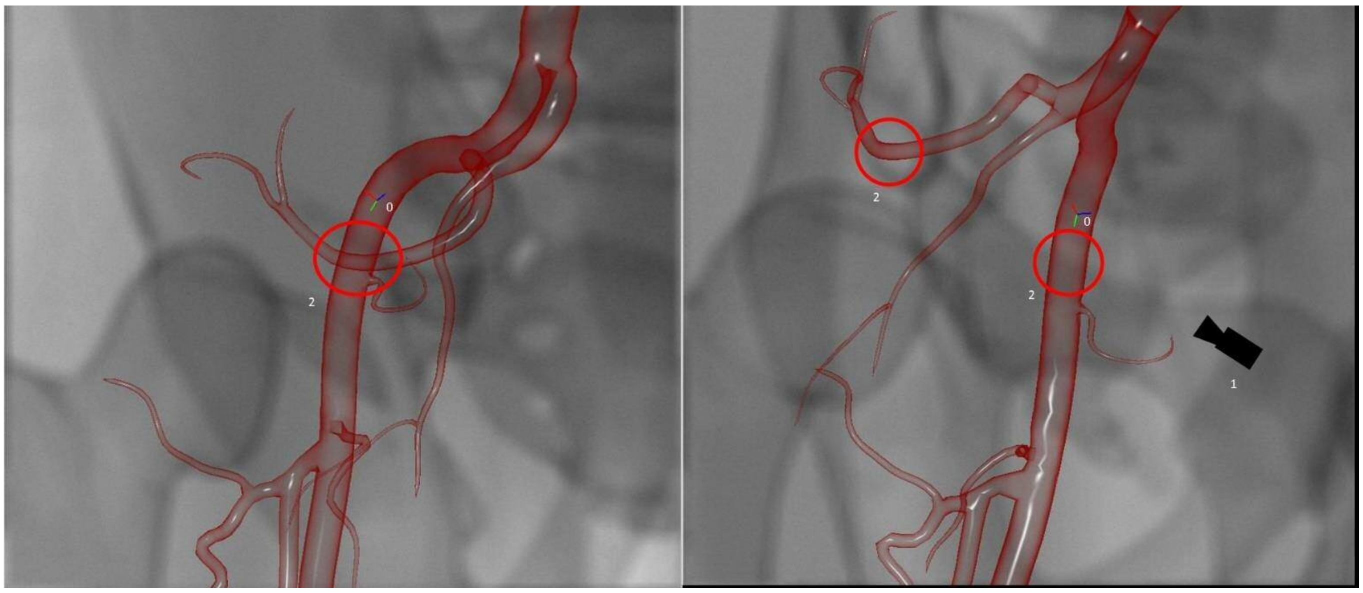


Figure 8. The geometry of the blurring artifact

0 - place of injection, direction of axes

1 - camera position

2 - overlaid areas creating an overlay artifact

The solution to this problem is based on the idea of filtering and interpolating values. It is necessary at the stage of the primary processing of the object, in the shader, when we still know the topology of the object to interpolate nonlinearly the vertex brightness values along the triangles. With this approach, we will get smooth transitions without steps in areas with different brightness. However, this solution does not solve all the problems. At the boundary of injection remains a sharp boundary, where the object of contrast ends. In a real situation, a cloud of contrast fluid is created at the injection site and the transition between the contrasted and the general part is not so sharp. To solve this problem, it is necessary to treat the injection site in a special way. It possible to draw the source object by applying a slightly modified interpolation approach. Thus, we get the image of the "cloud", which must be obtained with the main object of contrast.

You should not forget about all the artifacts already solved in the visualization of the main object.

With the help of the techniques presented above, most of the visualization of the contrast object is realized, but that's not all. It remains to visualize the effect of reflux and extravasate. Reflux is the effect when a small amount of contrast fluid under pressure breaks out of the main artery gate. Extravasate is a similar effect that arises from a rupture, an artery. Both these effects have a similar mechanics and look like a bundle of smoke escaping from the mainstream.

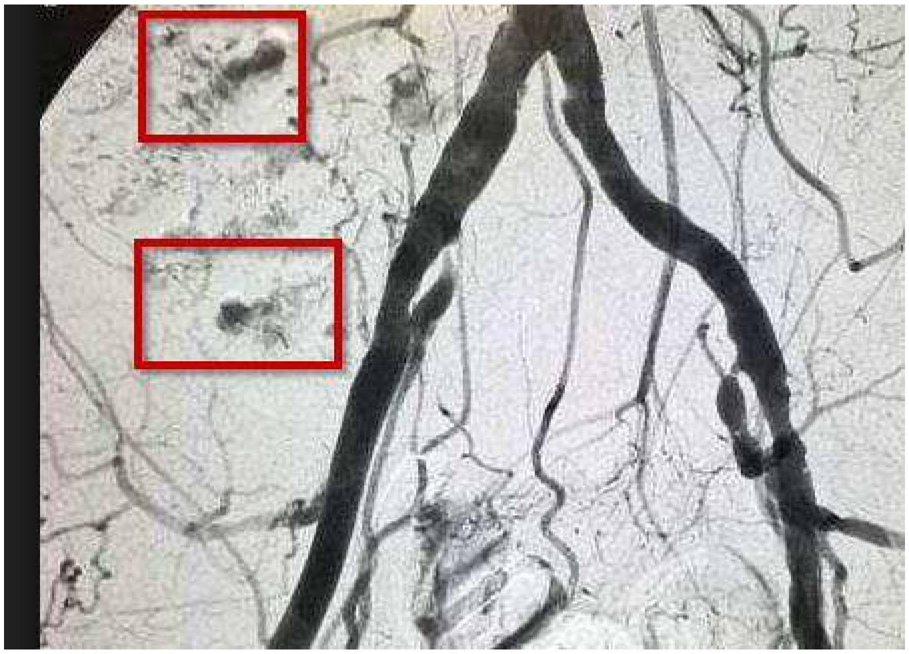


Figure 9. Extravasate and reflux smokes

The implementation of smoke is also a well-known computer graphics problem. The most frequent approach to solving it is simulation with particle systems. However, as already described above, such a resource-intensive solution does not suit us, so another approach was found that meets the schematic requirements. We use the sprite approach with some changes. Sprite itself is a flat object that always turns on the camera. Such a decision was widely used in early games and requires a minimum of resources for implementation, but the obvious drawback is its realism. If camera looks at the sprite object from a different angle, then its "plane" flashes into view. We will use a pre-calculated smoke sprite, which will rotate after the rotation of the vector that determines reflux and extravasate. In order that the smoke does not look flat, we will use a view from several angles that will seamlessly flow and complement each other. Such a seemingly primitive approach can nevertheless provide a good visual result. The smoke of extravasate itself is very foggy and does not have a clear structure, so when rotations and changes happen in space, if the sprites smoothly flow into each other, the human eye will not notice the deception. More details about implementation and results in the next chapter. Thus, using a variety of techniques, we covered the main problems and the algorithm is ready. The mathematical model of contrast does not require a lot of CPU resources, it is easy to implement. The visual part, though, is somewhat complex, but does not require large computational resources. This tells us that ultimately, for the visualization task, we were able to unload the CPU as much as possible - shifting the load to the video card, developing a special algorithm that takes into account the strengths of the shader computing.

2.1.2. Algorithm for visualization of blood

Blood simulation is one of the central modules of the laparoscopic simulator. Blood during the operation gives the surgeon important information about what is going on and the specific pathology of the patient. Above we have already described the concept of blood realization. The approach is that the blood by its behavior can be divided into 2 types: jet and surface. In this paper, we describe the algorithm of surface blood realization, the algorithm of the blood stream is described in [7].

By itself, the decomposition of the task of visualizing the blood into two different types already carries with it the simplification of calculations, but we will go further. The behavior of the surface blood is simulated by the movement of spots on the texture of the object through which it flows. Each particle - a drop of blood under the influence of gravity slowly drains down and accumulates in the cavities. On the basis of these particles, a texture must be collected, which will then be superimposed on the organ texture and drawn on the screen. Our algorithm must solve two main problems:

1) Movement of blood under the influence of gravity
2) The blood itself, being textural, must nevertheless visually behave like a convex object

In the process of computation, the algorithm should generate several textures. The first one is a diffuse texture. It will be superimposed on the object and thus the organ will stain. The second texture is the normal map. It is necessary for the calculation of highlights. Generating a normal map provides a solution to the second problem. Based on this texture, we can simulate the convexity, glare and the behavior of the rays of light when reflected from the blood. The main difficulty on the part of the algorithm is the implementation of the first problem. A drop of blood is a stain on the texture, it lives and moves along two-dimensional texture coordinates. We must somehow transmit information about the volume geometry of the organ into the texture space of the blood. The point on the texture should know in which direction it should move. It is necessary to convert from the global coordinates to the texture coordinates. In the world coordinates, the blood spot, while on the object, will move toward the projection of the gravity vector onto the plane of the object. Since the object consists of triangles, we must project the gravity vector in global coordinates to the triangle, obtaining the direction vector of the blood in global coordinates (Figure 10). Then this vector from the global coordinates must be translated into texture.

Let us consider in more detail the translation of the vector of the movement direction of blood from global coordinates to texture. This can be done with the help of barycentric coordinates known to us from the course of algebra and geometry [15]. Knowing the global and texture coordinates of the vertices in the triangle ABC and vector d it is not difficult to calculate the barycentric coordinates of vector d with respect to the triangle ABC. Multiplying the resulting barycentric coordinates by the texture coordinates of the vertices, we obtain the direction vector of the blood in the texture coordinates. This resulting vector is target of our calculation. Note that this procedure must be carried out for each triangle of the organ model, through which blood flows. Let`s pay attention, that the calculation inside each triangle does not depend on neighboring triangles and can be executed in parallel. The specificity of our application is that all calculations are carried out in real time, and the calculation of gravity is necessary for each piece of blood. However, the calculation of the direction of motion does not depend on the blood itself, but only on the geometry of the object. This pushes us to the idea that you we perform calculations at the start of the scene and then save the data for all the triangles. With the classical approach, solving this problem on the CPU, the data would be stored in a special

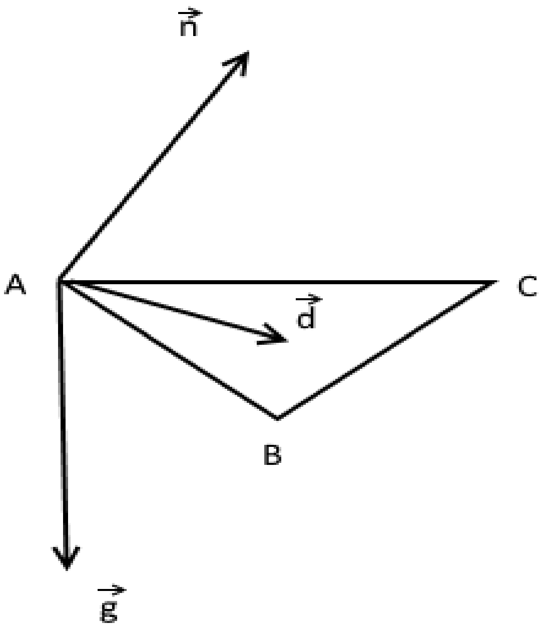


Figure 10. Gravity vector projection

ABC – one of the triangles of the model

– gravity vector

– normal vector to ABC

– direction vector of the blood in the plane of the triangle ABC

structure, the size of which would depend on the number of triangles in the model. However, we can generate a texture-scan of the object, where in each pixel the texture coordinates of the particular triangle in the color would contain the vector-direction of movement. With this approach, the space occupied does not depend on the size of the object but depends only on the texture resolution we have chosen, which we can adjust if necessary. Increasing the resolution - increasing the accuracy, reducing the resolution - reduce the amount of memory used.

The generation of a diffuse texture and a normal map does not represent a particular algorithmic complexity. Knowing the texture coordinates of the blood spots, you we generate a texture of the spots from these coordinates, blurring the boundaries. Note, however, that in general, the texture density of 3D objects is uneven, and this must be taken into account in the implementation. The task of generating such textures naturally falls on the architecture of the GPU.

2.2 Visualization algorithms of soft tissues

2.2.1 Visualization algorithms of coagulation

Coagulation is the process of transferring energy in order to increase the coagulability of blood. Most often this process is carried out by means of energy transfer with electricity. The tissue in this case, at the place of contact with the source, changes its structure - a coagulation spot appears. To implement the coagulation effect, we use an approach similar to the surface blood. We will generate a coagulation texture, which will be applied to the object of the organ. With this approach, we will obviously get all the advantages in terms of performance and data storage as in the case of texture blood. For the logic of surgery, it is important to know where the coagulation was performed. Coagulated tissues do not bleed, have other physical properties. The surgeon performing the operation should clearly follow the instructions, coagulate only in the necessary places. For the implementation of the logic operations we need to know specifically where the coagulation was. Having received such a texture, we actually close the question with the logic of coagulation. Simulation logic modules can simply refer to such a texture, take the pixel value by coordinates. In a pixel the normalized value of energy reported to this point of body is stored.

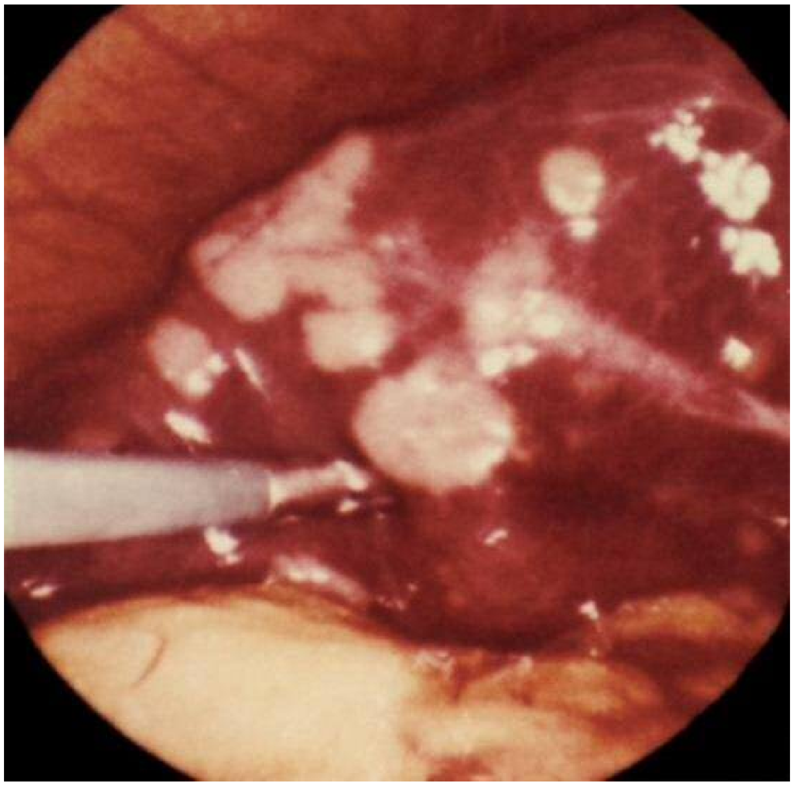


Figure 11. Coagulation spots

Electrocoagulation can be of two types: monopolar and bipolar. In case of monopolar coagulation, the energy is distributed in a certain radius around the point of contact of the organ and the instrument. In bipolar the impact of the current passes between the electrodes and affects only the part of the body, which is captured by the tool.

Generation of textures with monopolar effects is based on the distribution of the patches around the points of contact. The generator takes as input the texture coordinates of the points of contact. Then, in the Shader stage, the neighboring pixels color themselves, the radius at which the neighboring pixels are affected depends on the power of the transmitted energy. After that, additional visual effects are applied: such as special noise, texture blending, etc. In case of bipolar influence on the input program is fed vertices captured by the tool. Another object is selected from the main three-dimensional object, which includes only triangles containing vertices captured by the tool. This object is then drawn on the texture of the sweep. Note that when selecting the captured object, the topology of the source object of the body must be taken into account. The fact is that so-called "connecting" triangles can be formed during the simulation. These are triangles with vertices on the texture scan lying at a great distance. It is necessary to properly handle such triangles, since painting it completely we get not an objective picture of coagulation. A similar situation may also appear when you capture some distant from each other parts in the folds.

2.2.2 General and final visualization of organs, conclusion of the chapter

Generation of a texture of blood, normal maps of the blood texture coagulation are ultimately required for visualization of soft tissues of human internal organs. Just by drawing a kidney with a texture on it, you can't have a medical simulation. Module operation includes a logic operation and logic, simulating common physiological processes. Blood must be visible on the objects, coagulation must change the characteristics of the organs, the simulator must display an interactive picture, which depends on the specific actions of the surgeon. All textures and data prepared in the simulation process are applied at the last stage of drawing the internal organs.

The architecture of graphical cards and shader calculations on them are designed to work with images and generate textures. The narrow specifics and limitations allow the GPU to generate a huge number of threads. Development of algorithms with these features can save the necessary resources. It would seem that only a small range of problems can be solved in this way, but we have shown in this Chapter that with the right approach, it is possible not only to speed up calculations and unload the CPU, but also to organize the storage and data structure more efficiently and flexibly.

Chapter 3

Software implementation of algorithms for visualization of internal human tissues

3.1 Implementation of algorithms for visualization of liquid tissues

Modern medical simulations produce a lot of calculations. It is necessary to work with a large amount of data in real time. For this reason, the main language is c ++, as it is very efficient and suitable for this task best. Programs, the result of which is presented in this chapter are implemented on a special graphics engine based on DirectX. This graphics engine provides management of global resources, parameter transmission and high-level logic. Writing code for this system requires special skills and knowledge, which is not the goal of our work. So, the source code is stored in binary encrypted form, the software itself is proprietary, and the reader cannot run it. However, the executable code for the CPU is written in standard c ++, shaders in the HLSL language. Global interaction between shaders and modules with c++ is presented in the work in the form of block diagrams. The code of the basic shaders and programs is presented in the Appendix. In this chapter, key aspects of the implementation of algorithms will be analyzed.

3.1.1 Implementation of algorithms visualization of contrast fluid

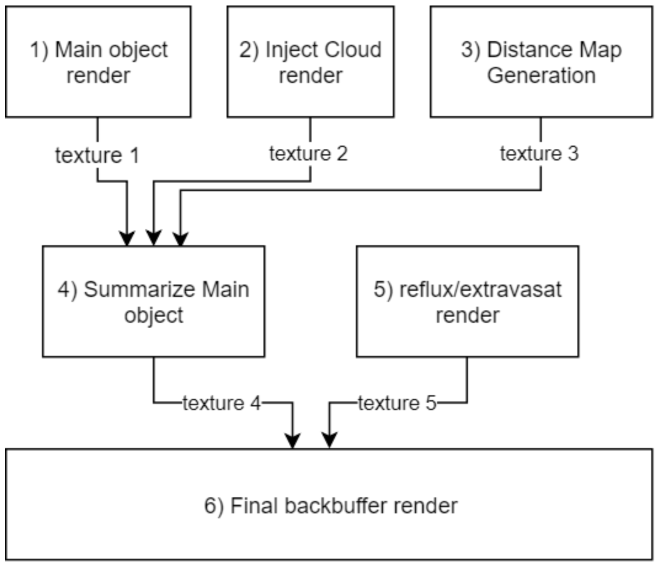


Figure 12. Flowchart for visualization of contrast fluid motion

From the previous Chapter, we remember that the image of contrast is formed in several stages. Each stage provides one or more passes of the corresponding shader. The result of each step is a texture that will be used in the next step. Figure 12 is a block diagram describing the implementation of the algorithm for visualizing the movement of contrast fluid in the simulator of endovascular intervention.

1) Main Object Render [appendix 2.a] – at this stage, the main object is rendered into the texture. Overlap artifacts are taken into account. Blurring occurs by interpolating the brightness along the triangle. The image is monochrome and for the transmission of brightness, you can use only the red channel in order to save memory and resources. In the process of this shader, texture 1 is generated. (Figure 13) Note that in Figure 13, unlike Figure 3, there are no clear boundaries between

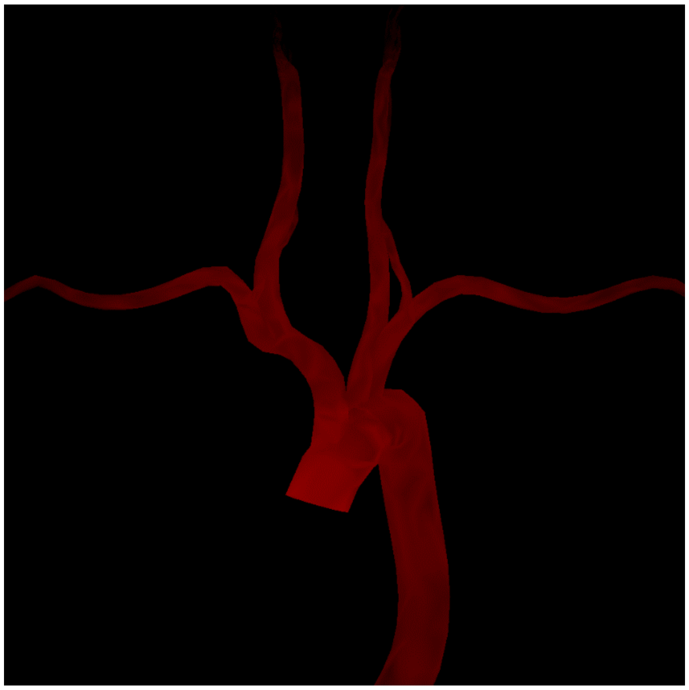


Figure 13. Texture 1.

neighboring polygons, and the overlap problem is also solved.

2) Inject Cloud render [appendix 2.b] – here the object of "cloud" contrast is rendered at the injection site. A texture 2 is generated (figure 14) which then needs to be applied to texture 1 using the information from texture 3 (figure 15).

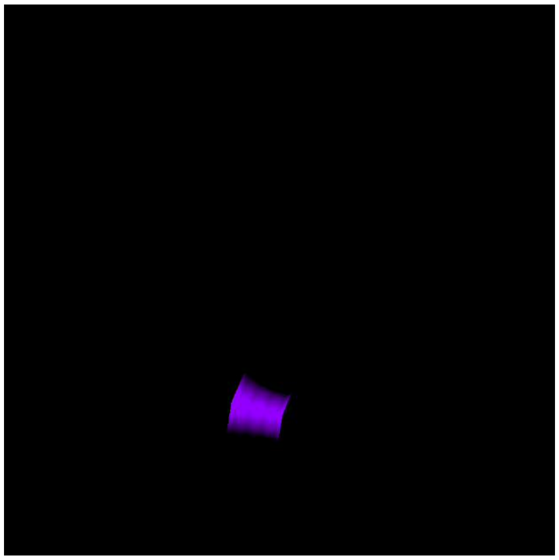


Figure 14. Texture 2. Contrast "Cloud" at the injection point

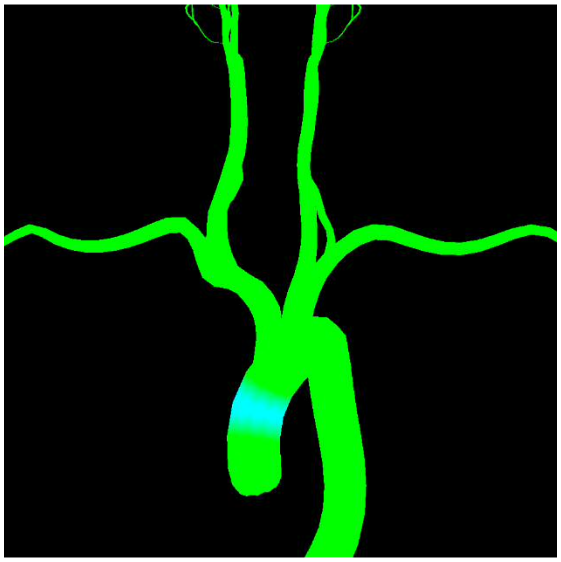


Figure 15. Texture 3. Distance map to injection point

3) Distance Map generation [appendix 2.c] – At this stage, a distance map from the injection site is generated (Figure 15). Green color fill the pixels that are at a great distance from the injection site. Cyan turning to green - the normalized distance to the injection site.

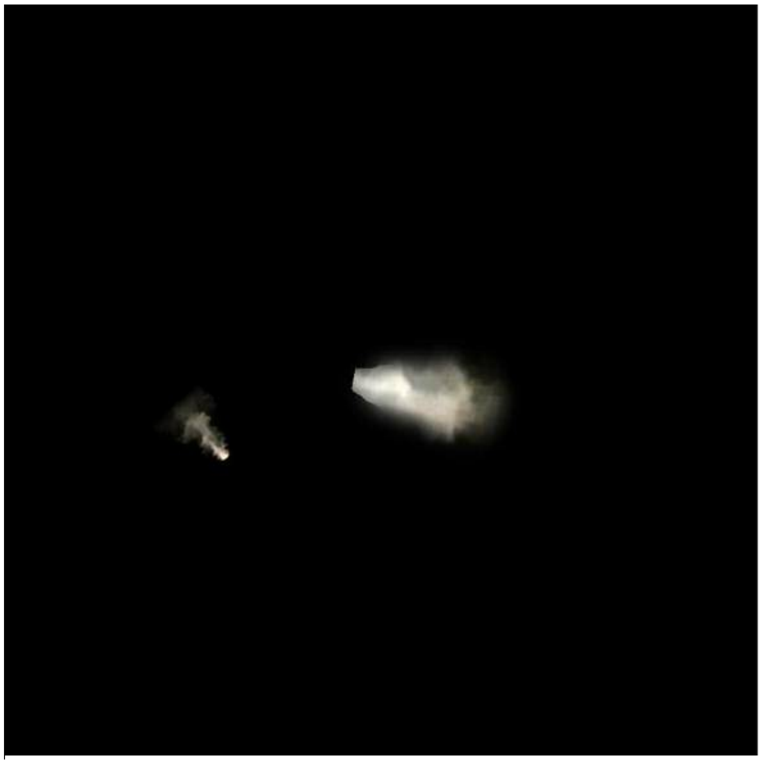


Figure 16. Texture 5. Reflux and extravasate

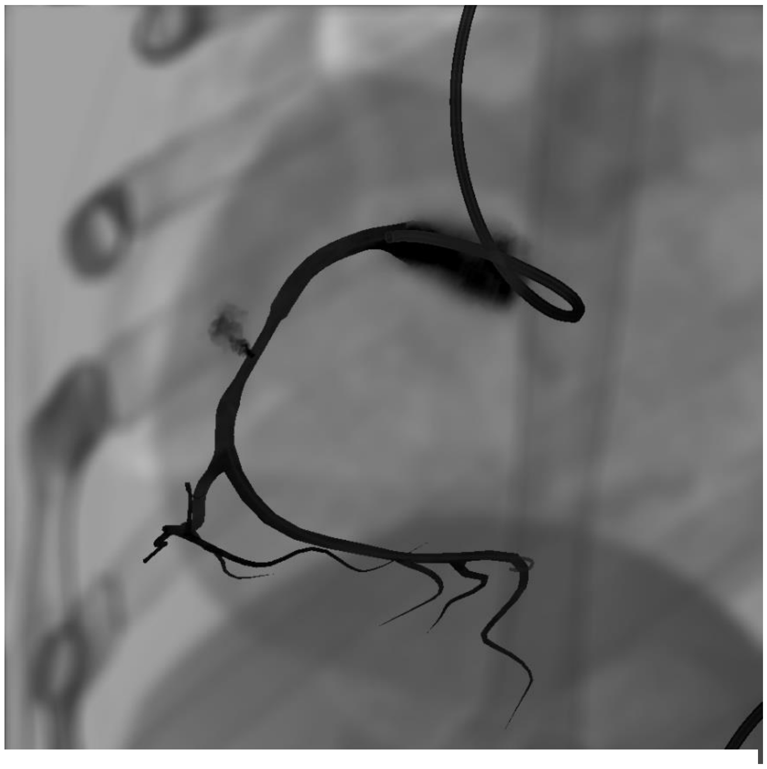


Figure 17. Result screenshot

4) Summarize Main object [appendix 2.d] - on the basis of the previous three stages, an image is generated that combines a cloud of contrast. Use a distance map to ensure a smooth transition between images.

5) Reflux/extravasat render [appendix 2.e] – smoke sprite generation. The final sprite is obtained by combining and mixing sprites obtained from different points of view. A texture 5 is generated (figure 16) For more obvious visualization, a frame is taken from another location inside the vessels. Sprites are rotated behind the direction vector in screen-space coordinates. Smoothing change of projections is calculated.

6) Final backbuffer render [appendix 2.f] – The final processing of all textures and rendering the result to the screen. At

this stage, the final image is formed, all the textures are collected in one and fed to the screen. Figure 17 shows the result of the program. It can be seen 2 hazes. A small smoke in the left part-extravasate, obtained by rupture of the vessel in this place. Big smoke to the right, reflux, escaping from the gates of the vessel in the large artery. Figure 18 shows the result of the algorithm in the first case. Thus, we implemented the algorithm proposed in Chapter 2. Took into account all potential problems, excluded artifacts found in the analysis of the problem. The main load of the algorithm implementation, the calculations are on the shader calculations of the graphics card. This means that our goal to unload the CPU is achieved.

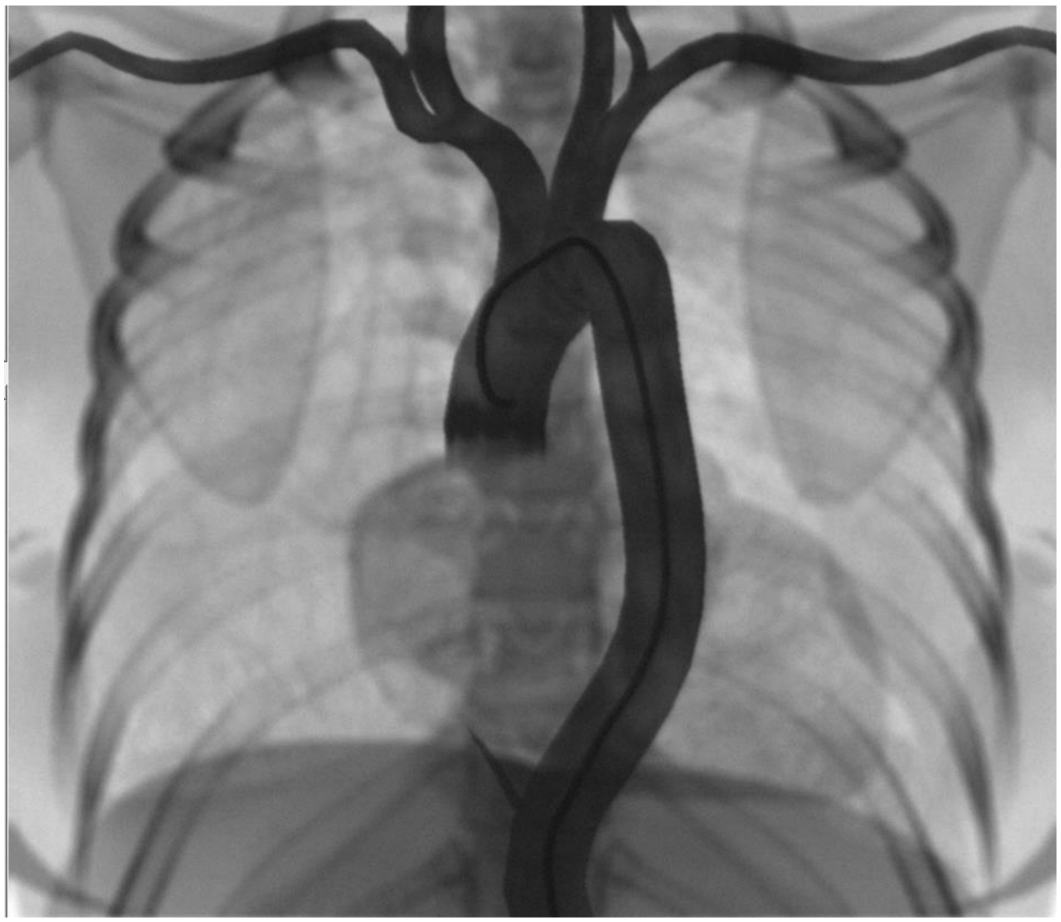


Figure 18. Contrast Final screenshot

3.1.2. Implementation of algorithms visualization of blood.

The calculation of the direction of blood movement is based on the transition from the world coordinates to the texture ones. On each particle of blood gravity acts and we need to determine in which direction the texture of the blood drop should move. We solved this problem by using the transition through the barycentric coordinates. Below is a function that calculates barycentric coordinates.

```
void BpGravity::Barycentric(D3DXVECTOR3 a, D3DXVECTOR3 b, D3DXVECTOR3 c,
D3DXVECTOR3 p, float &u, float &v, float &w)
{
    D3DXVECTOR3 v0 = b - a;
    D3DXVECTOR3 v1 = c - a;
    D3DXVECTOR3 v2 = p - a;

    float d00 = DotProduct(v0, v0);
    float d01 = DotProduct(v0, v1);
    float d11 = DotProduct(v1, v1);
    float d20 = DotProduct(v2, v0);
    float d21 = DotProduct(v2, v1);
    float denom = d00 * d11 - d01 * d01;
    v = (d11 * d20 - d01 * d21) / denom;
    w = (d00 * d21 - d01 * d20) / denom;
    u = 1.0f - v - w;
}
```

The procedure takes as the input 4 coordinates in the world space. Vectors a,b, c are the coordinates of the triangle, respectively. The vector p is the coordinates of the direction vector ABC triangle space. u,v, w – calculated barycentric coordinates. Using this function, we can easily move from world coordinates to texture. This function implements the primitive solution of the linear equation according to the formula of Kramer. This procedure is performed for each polygon in the model you are looking for when the scene is loaded. The procedure for processing the landfill is presented in [appendix 2.g]. Note that it is called in parallel and is thread-safe.

Figure 19 shows the result of the program, which consists in the generation of two textures of blood: diffuse and normal maps. Diffuse texture contains color and shape. Based on the map of normals, glare and illumination of the object are calculated, which creates a volume effect.

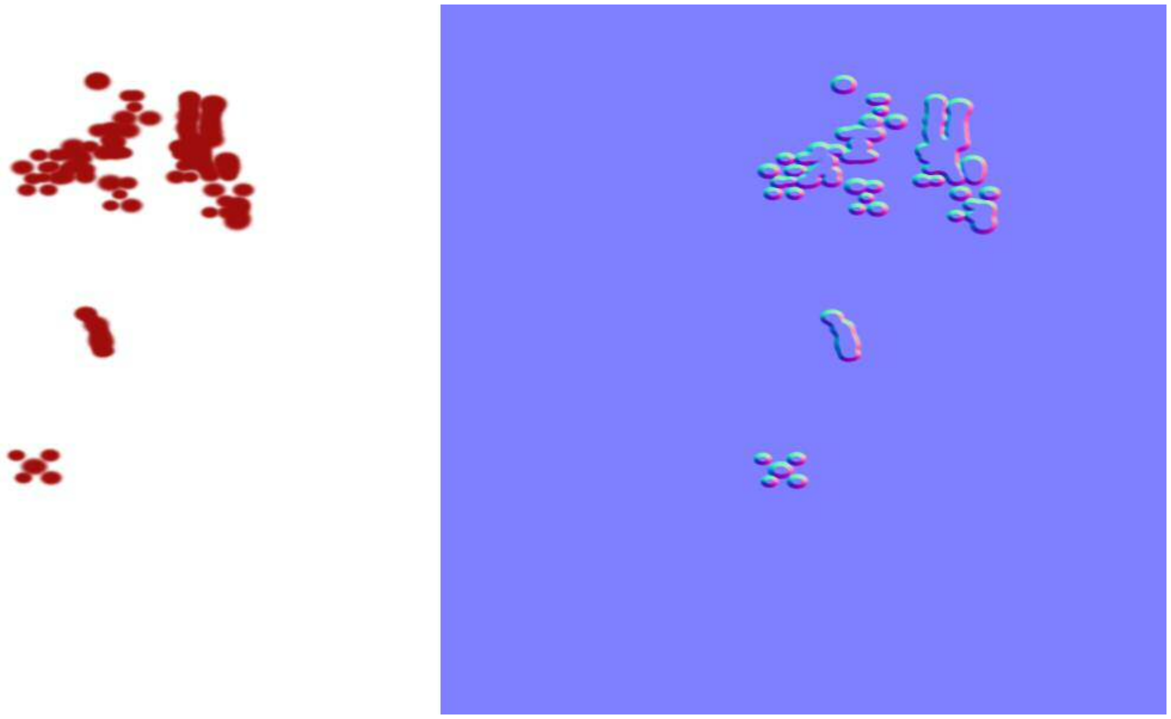


Figure 19. Diffuse texture and normal map of the surface blood

3.2 Implementation of soft tissue visualization algorithms

3.2.1. Implementation of the algorithm of coagulated tissues

Bipolar and monopolar coagulation are actually two different types of coagulation. The approach to the solution is different and based on different principles, however in both cases a texture of the transmitted energy is generated. Below is a block diagram describing the implementation of the coagulation algorithm.

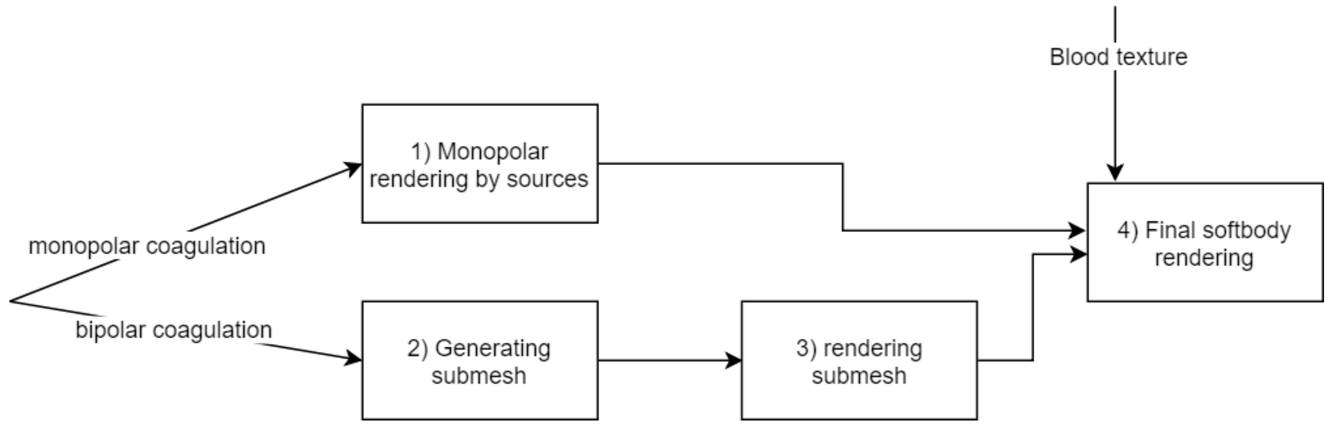


Figure 20. The block diagram of algorithm of coagulation

1) Monopolar rendering by sources [appendix 2.h] – Drawing of spots on the basis of points of contact. Neighboring pixels around the input points are colored according to the radiation power and the duration of the exposure.
2) Generating submesh [appendix 2.j] – the selection of a submesh from the object based on the vertices captured by the coagulation tool. The received three-dimensional object is transferred to the next stage.

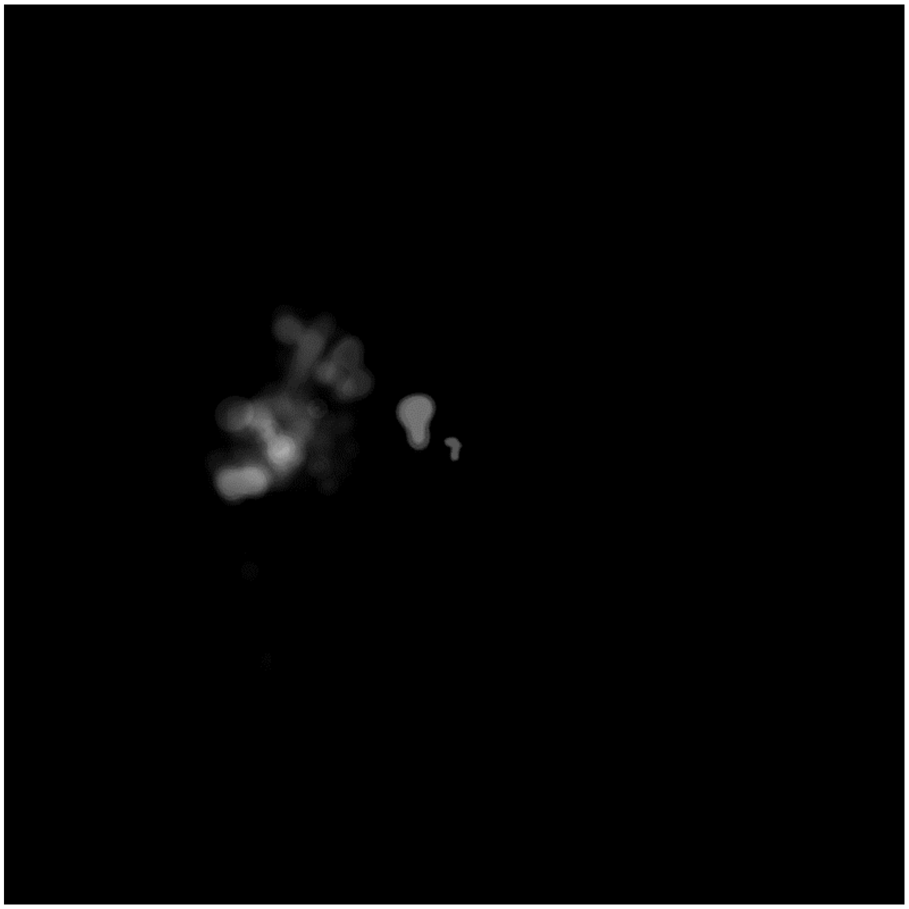


Figure 21. Energy texture

3) Submesh rendering [appendix 2.k] – the selected object is rendered on the energy texture. The texture of the coagulation is ready.

In Figure 21, you can see the result of the algorithm in the form of a texture of the transferred energy. White spots on a black background show those places where the electrodes are in contact with the organ. The resulting texture, along with the textures, obtained in the blood algorithm is transferred to the final shader.

3.2.2 General implementation of the final shader of the soft bodies

Algorithms of coagulation and blood in the process of their work prepare data for the last stage - the visualization of organs. These processes affect the physical and visual properties of objects. Ultimately, the goal of medical simulation is to reflect the behavior of internal organs, which is impossible without the aforementioned algorithms. Plus, the classic rendering of soft bodies in medical simulation adds these effects. At the last stage of the image formation, at the stage of texture blending, blood and coagulation effects are superimposed on the organ. Blood voluminous appears because of the generated normal map, glare is calculated on its basis, which adds a presence effect. Coagulation also visually changes the structure of the object, gives it a haze and realistically highlights the impact sites. The shader itself is presented in [Appendix 2.m] Figures 22 and 23 show the results of the imaging of organs using blood and coagulation effects, respectively

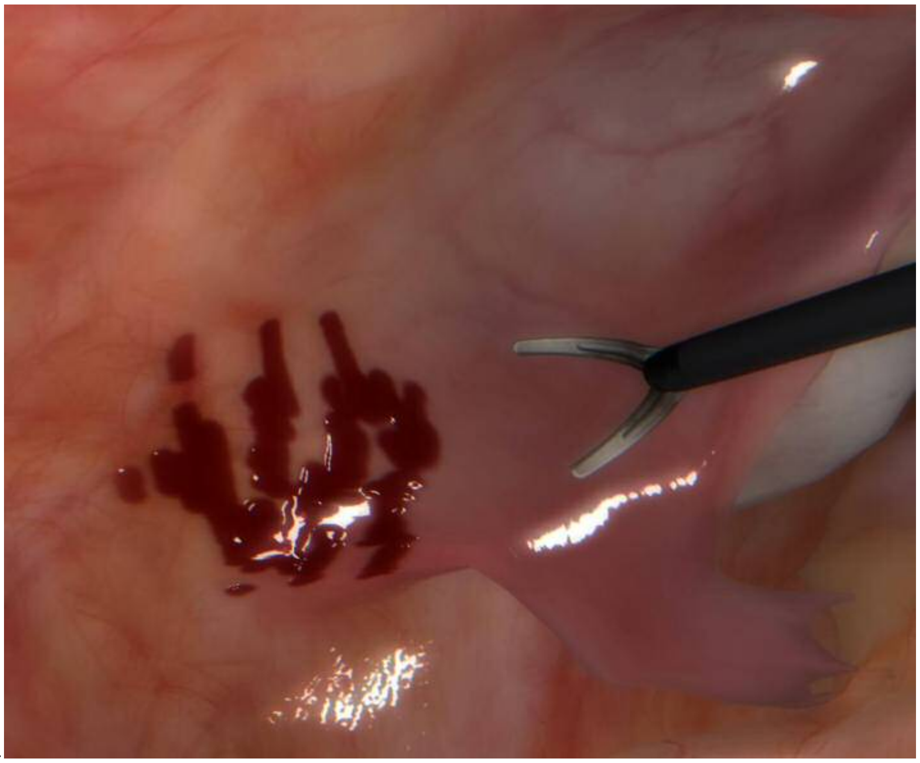


Figure 22. Visualization of human soft tissues with blood spots

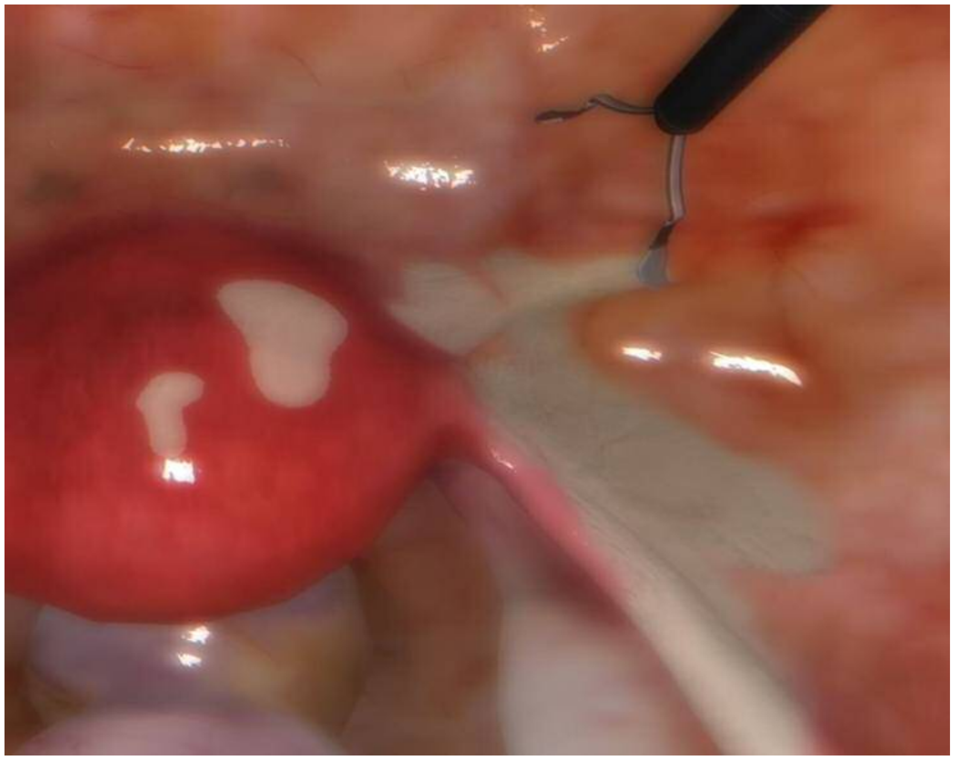


Figure 23. Visualization of human soft tissues with traces of coagulation

Conclusion

In this diploma thesis, the goals were achieved, namely: algorithms for visualization of soft and liquid tissues of the human body in medical simulations. The algorithms are constructed in such a way that the main load falls on the calculations on the graphics card. All algorithms take no more than 3ms of computing time within one frame. We have shown that the development of such algorithms is important and can increase the efficiency of the software as a whole. The implementation of the objectives of General purpose shaders not only allows us to quickly and efficiently generate texture. We have shown that using this approach, we can also more effectively organize the storage of data, and some logical functions. The approach proposed in this work covers a wider part than the original scope of shaders. To use all the advantages, it is necessary to analyze ways of solving problems in a slightly different way, taking into account the strengths of the device.

These algorithms and programs can certainly be improved. The main vector of development of such algorithms and the entire simulation medicine lies in the direction of rehearsals of operations. The algorithms should take the data of real patients and process them automatically without human intervention and information prepared manually. The complexity of such data is clear: high noise, low resolution and overall quality.

The work is certainly of great social importance, because the programs and algorithms presented in it are used in the teaching medical software. Our lives depend on the quality of training of doctors.

Bibliography

[1] Harris, Mark J. 2004. "Fast Fluid Dynamics Simulation on the GPU." In GPU Gems, edited by Randima Fernando, pp. 637–665. Addison-Wesley.

[2] Müller, Matthias, David Charypar, and Markus Gross. 2003. "Particle-Based Fluid Simulation for Interactive Applications." In Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp. 154–159.

[3] JOHANSON, C. 2004. Real-time water rendering - introducing the projected grid concept. Master's thesis, Department of Computer Science, Lund University.

[4] LIU, G. R., AND LIU, M. B. 2003. Smoothed Particle Hydrodynamics: A Meshfree Particle Method. World Scientific.

[5] LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. SIGGRAPH Comput. Graph. 21, 4, 163–169.

[6] ROSENBERG, I. D., AND BIRDWELL, K. 2008. Real-time particle isosurface extraction. In SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games, ACM, New York, NY, USA, 35–43.

[7] Nikolay V. Spiridonov , 2015. Real-Time Rendering of Dynamic Fluid Jets. Research Journal of Applied Sciences, 10: 436-441

[8] https://en.wikipedia.org/wiki/Interventional_radiology#Vascular

[9] http://www.gamedev.ru/code/articles/HLSL

[10] https://en.wikipedia.org/wiki/High-Level_Shading_Language

[11] https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx

[12] https://msdn.microsoft.com/en-us/library/windows/desktop/bb172517(v=vs.85).aspx

[13] https://en.wikipedia.org/wiki/Painter%27s_algorithm

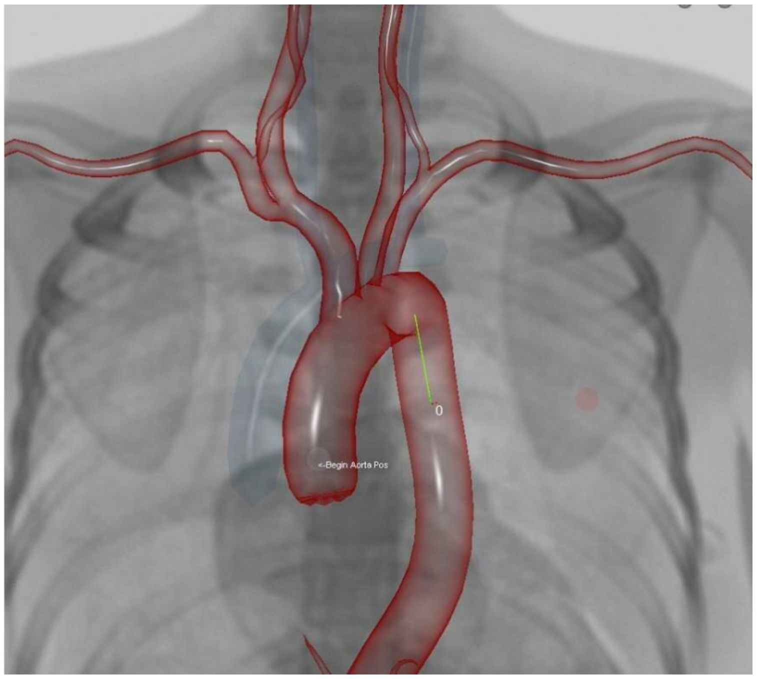
[14] Foley, James; Feiner, Steven K.; Hughes, John F. (1990). Computer Graphics: Principles and Practice. Reading, MA, USA: Addison-Wesley. p. 1174. ISBN 0-201-12110-7.

[15] https://en.wikipedia.org/wiki/Barycentric_coordinate_system

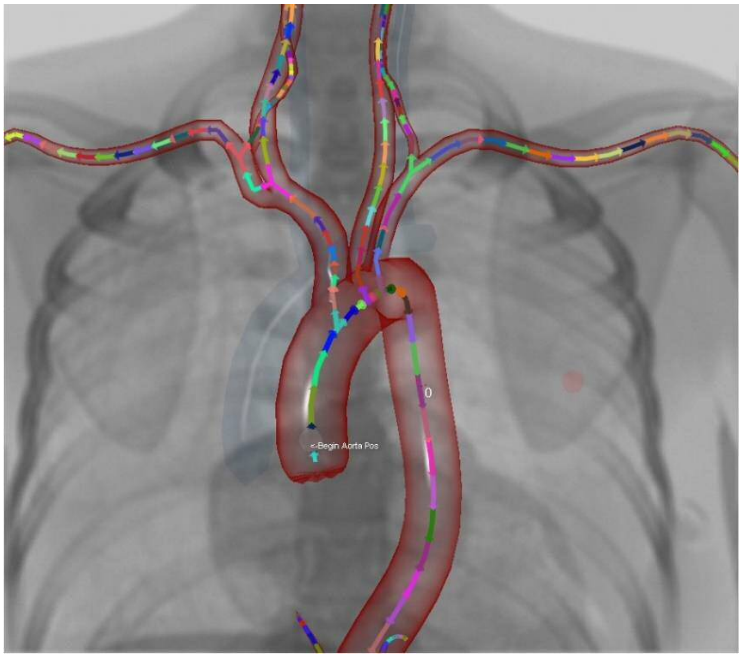
Appendix

1. Figures

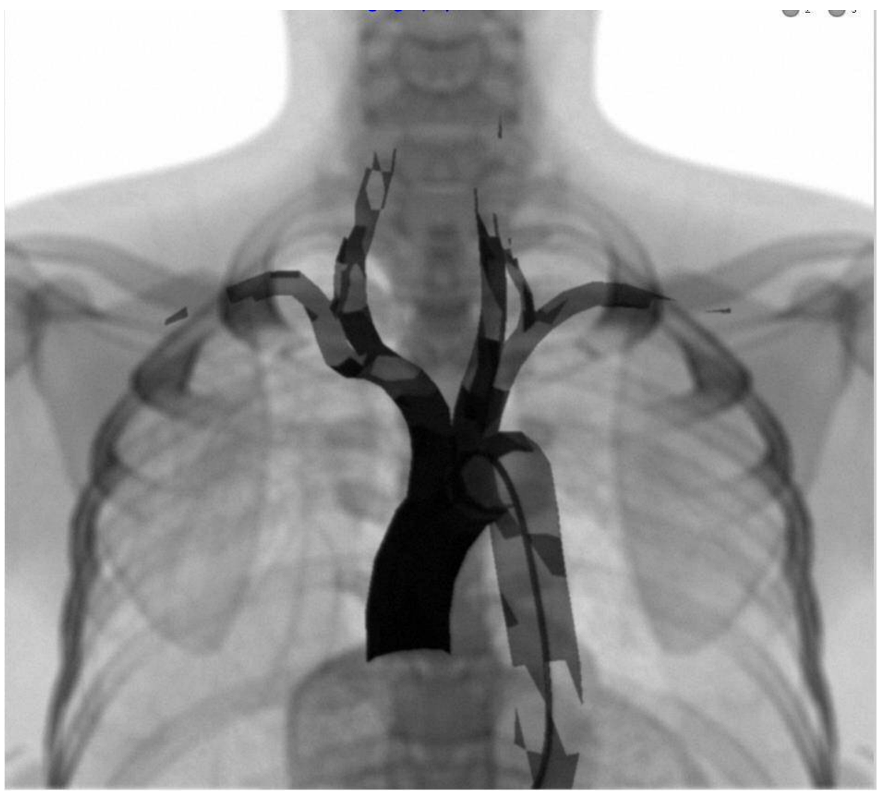
a. Source model of vessels



b. Graph of blood flow



c. Painted object that is returned from a logic module of the contrast in the visualization module



2. Source code

c. Main object render

```
//Matrices definition
float4x4 WorldVP : WorldViewProjection;     // The multiplication of the world, view and projection
matrices
float4x4 mWorld : World;                    // The world transformation matrix
float4x4 mWorldIT : WorldInverseTranspose;  // Inverse - transpose of the world matrix
float4x4 ViewInv : ViewInverse;             // Inverted matrix screen

float3  SpecularColor : CHANNELVECTOR0;

//samplers for texture
texture sourceTextureMap :  TEXTURE0;
sampler2D sourceMapSampler = sampler_state
{
    Texture = <sourceTextureMap>;
    MinFilter = Anisotropic;
    MagFilter = Anisotropic;
    MipFilter = Anisotropic;
};

/////////////////////////////////////////////////////////////////////////
///// Structures ////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////

// Input Data
struct appData
{
    float4 position     : POSITION;         // Object position
    float3 Normal       : NORMAL;           // Normal to vertex
    float2 texcoord0    : TEXCOORD0;        // Texture coordinates ID=1
    float2 texcoord1    : TEXCOORD1;        // Texture coordinates ID=2
    float2 texcoord2    : TEXCOORD2;        // Texture coordinates ID=3
    float4 color        : COLOR;

};

// Output Data
struct vertexOutput
{
    float4 HPosition    : POSITION;
    float2  UV          : TEXCOORD0;
    float4  color       : TEXCOORD1;
    float3 Normal       : NORMAL;
};

struct pixelOutput
{
    float4 col          : COLOR;
    float dep           : DEPTH;
};

//////////////////////////////////////////////////////////////////////////
///// Shaders ////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

// Vertex шейдер
vertexOutput VS(appData IN)
{
    vertexOutput OUT;

    float4 p = mul(IN.position,WorldVP);
    const float3 nullPos = float3(0,0,0);

    p.xyz = lerp(p.xyz, nullPos,0.001*IN.color.a);

    OUT.HPosition = p;
    OUT.UV = IN.texcoord0;
    // contrast brightness inside the color
    OUT.color = IN.color;
    OUT.Normal = IN.Normal;

    return OUT;
}

pixelOutput PS(vertexOutput IN)
{
    pixelOutput res;

    float4 tex =  tex2D(sourceMapSampler ,  IN.UV);

    res.col.rgb = SpecularColor;
    res.col.a = 0;

    res.col.rgb *= IN.color.a;

    const float deegreeOfSmoothing = 25;
    for(int i = 0;i < deegreeOfSmoothing; i++)
    {
        float   ratio = i / deegreeOfSmoothing;
        float x = IN.color.a - ratio;
        res.col.a += saturate( (1 / (( 1-ratio ) / 2)) * x);
    }

    res.dep  = 1 - IN.color.a;
    return res;
}

////////////////////////////////////////////////////////////////////////
///// Techniques///////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

technique render_cw_culling
{
    pass p0
    {
        CullMode=none;

        AlphaBlendEnable = true;
        ALPHAFUNC = 0;

        ZENABLE = 1;
        ZWriteEnable = true;
        ZFunc = 4;

        VertexShader = compile vs_3_0 VS();
        PixelShader = compile ps_3_0 PS();
    }
}
```

b. Inject cloud render

```
// Declaring arrays
float4x4 WorldVP: WorldViewProjection;     // Work World, species and projection matrices
float4x4 mWorld: World;                    // world transformation matrix
float4x4 mWorldIT: WorldInverseTranspose;  // Inverted - transpose of the world matrix
float4x4 ViewInv: ViewInverse;             // Invert screen matrix


float   radius: CHANNELVALUE0;
floatbrightness: CHANNELVALUE1;
floatbriRatio: CHANNELVALUE2;


float3SpecularColor: CHANNELVECTOR0;
float4  injectPos: CHANNELVECTOR1;



////////////////////////////////////////////////// //////////////////////
///// Structure /////////////////////////////////////////// ////////////
////////////////////////////////////////////////// //////////////////////

// Incoming data
struct appData
{
float4 position: POSITION;         // Position the object
```

```
float3 Normal: NORMAL;           // The normal to the surface
float2 texcoord0: TEXCOORD0;        // Texture coordinates ID = 1
float2 texcoord1: TEXCOORD1;        // Texture coordinates ID = 2
float2 texcoord2: TEXCOORD2;        // Texture coordinates ID = 3

float4 color: COLOR;

};

// Outgoing data
struct vertexOutput
{
float4 HPosition: POSITION;         // Position the object
float2  UV: TEXCOORD0;
float4  color: TEXCOORD1;
float4 wPosition: TEXCOORD2;
float4 sPos: TEXCOORD3;

};

struct pixelOutput
{
float4 col: COLOR;
float dep: DEPTH;
};


////////////////////////////////////////////////// //////////////////////
///// Shaders ////////////////////////////////////////// //////////////
////////////////////////////////////////////////// //////////////////////

// Vertex shader
vertexOutput VS (appData IN)
{
    vertexOutput OUT; // Reset output structure

float4 p = IN.position;
float3 CamPos = float3(0.0.0);

OUT.HPosition = mul (p, WorldVP);
OUT.wPosition = mul (p, mWorld);
OUT.sPos = mul (p, WorldVP);

OUT.UV = IN.texcoord0;
OUT.color = IN.color;

return OUT;
}


pixelOutput PS (vertexOutput IN)
{
pixelOutput res;

float4 pos = IN.wPosition;

float halfr = radius /3;
float distToInject = length (pos - injectPos);

res.col.rgb = SpecularColor;
    res.col.a = 1;

res.col.rgb * = brightness /255;
res.col.rgb * = (float) (DistToInject <radius);

float = Ratio - (1/ Halfr) * distToInject +3;

res.col.rgb = lerp (float3(0.0.0), Res.col.rgb, saturate (ratio));
res.col.b = saturate (ratio);
res.col.a = (float) ((Ratio)>0);


res.col.a * = (float) (Res.col.r> = briRatio);

res.dep = res.col.a;

return res;
}


////////////////////////////////////////////////// //////////////////////
///// Equipment ////////////////////////////////////////////// //////////////
////////////////////////////////////////////////// //////////////////////


technique render_cw_culling
{
pass p0
{

CullMode = ccw;

        ZWriteEnable = true;
        AlphaBlendEnable = true;
        ZFunc = ALWAYS;

        ZENABLE = 1;
        ZWriteEnable = true;
        ZFunc = 4;

        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS ();

}

}
```

c. Distance Map Generation

```
// Declaring arrays
float4x4 WorldVP: WorldViewProjection;     // Work World, species and projection matrices
float4x4 mWorld: World;                    // world transformation matrix
float4x4 mWorldIT: WorldInverseTranspose;  // Inverted - transpose of the world matrix
float4x4 ViewInv: ViewInverse;             // Invert screen matrix


float   radius: CHANNELVALUE0;
floatbrightness: CHANNELVALUE1;


float3SpecularColor: CHANNELVECTOR0;
float4  injectPos: CHANNELVECTOR1;


float2 ScreenToUV (float2 suv)
{
```

```
suv.y = -suv.y;
return (Suv +1) *0.5;
}

////////////////////////////////////////////////// //////////////////////
///// Structure ////////////////////////////////////////// ////////////
////////////////////////////////////////////////// //////////////////////

// Incoming data
struct appData
{
float4 position: POSITION;          // Position the object
float3 Normal: NORMAL;            // The normal to the surface
float2 texcoord0: TEXCOORD0;        // Texture coordinates ID = 1
float2 texcoord1: TEXCOORD1;        // Texture coordinates ID = 2
float2 texcoord2: TEXCOORD2;        // Texture coordinates ID = 3

float4 color: COLOR;

};

// Outgoing data
struct vertexOutput
{
float4 HPosition: POSITION;         // Position the object
float2  UV: TEXCOORD0;
float4  color: TEXCOORD1;
float4 wPosition: TEXCOORD2;
float4 sPos: TEXCOORD3;

};


////////////////////////////////////////////////// //////////////////////
///// Shaders ////////////////////////////////////////// //////////////
////////////////////////////////////////////////// //////////////////////

// Vertex shader
vertexOutput VS (appData IN)
{
    vertexOutput OUT; // Reset output structure

float4 p = IN.position;
float3 CamPos = float3(0.0.0);

OUT.HPosition = mul (p, WorldVP);
OUT.wPosition = mul (p, mWorld);
OUT.sPos = mul (p, WorldVP);

OUT.UV = IN.texcoord0;
OUT.color = IN.color;

return OUT;
}


float4 PS (vertexOutput IN): COLOR
{
float2 suv = ScreenToUV (IN.sPos);


float4 pos = IN.wPosition;
float4 res;

float halfr = radius /3;
float distToInject = length (pos - injectPos);

// float b = (float) (distToInject> radius);
float b = - (1/ Halfr) * distToInject +3;

    res = float4(0.1, Saturate (b), 1);

return res;
}


////////////////////////////////////////////////// ///////////////////////
///// Equipment ////////////////////////////////////////////// ///////////////
////////////////////////////////////////////////// ///////////////////////


technique render_cw_culling
{
pass p0
{

CullMode = ccw;

        ZWriteEnable = true;
        AlphaBlendEnable = true;
        ZFunc = ALWAYS;

        VertexShader = compile vs_3_0 VS ();
PixelShader =compile ps_3_0 PS ();

}

}
```

d. Summarize main object

```
// Declaring arrays
float4x4 WorldVP: WorldViewProjection;     // Work World, species and projection matrices
float4x4 mWorld: World;                    // world transformation matrix
float4x4 mWorldIT: WorldInverseTranspose;  // Inverted - transpose of the world matrix
float4x4 ViewInv: ViewInverse;             // Invert screen matrix


float   radius: CHANNELVALUE0;


float3  injectsuv: CHANNELVECTOR0;

// samplers for texture
texture sourceTextureMap: TEXTURE0;
sampler2D sourceMapSampler = sampler_state
{
Texture = <sourceTextureMap>;
    MinFilter = Anisotropic;
    MagFilter = Anisotropic;
    MipFilter = Anisotropic;
};
texture cloudMap: TEXTURE1;
sampler2D cloudSampler = sampler_state
{
Texture = <cloudMap>;
// anntialiazing
    MinFilter = Anisotropic;
    MipFilter = Anisotropic;
    MagFilter = Anisotropic;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};

texture distanceMap: TEXTURE2;
sampler2D distanceSampler = sampler_state
{
Texture = <distanceMap>;
// anntialiazing
    MinFilter = Anisotropic;
    MipFilter = Anisotropic;
    MagFilter = Anisotropic;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};


// IO structures
struct appdata {
    float4 Position: POSITION;
    float2 UV: TEXCOORD0;
};

struct vertexOutput {
float4 HPosition: POSITION;
float2  UV: TEXCOORD0;

};

vertexOutput VS (appdata IN)
{
vertexOutput OUT;
OUT.HPosition = IN.Position;
OUT.UV = IN.UV;


float4 p = IN.Position;
float4 wPos = mul (p, mWorld); // Get the vertex position in world space

return OUT;
}

float4 PS (vertexOutput IN): COLOR
{
```

```
injectsuv.y = -injectsuv.y;
injectsuv.xy = (injectsuv.xy +1) /2;

float4srcPixel = tex2D (sourceMapSampler, IN.UV);
float4 cloudPixel = tex2D (cloudSampler, IN.UV);
float4distancePixel = tex2D (distanceSampler, IN.UV);

float4 res = srcPixel;

res.r = lerp (srcPixel.r, cloudPixel.r, cloudPixel.b);

if(DistancePixel.b! = 0 && srcPixel.r * 0.9 > Res.r)
        res.r = (res.r + srcPixel.r) / 2;

// res.r = max (srcPixel.r, res.r);

return res;

}

technique HBlur
{
pass p0
{
CullMode = ccw;
                AlphaBlendEnable = false;
                VertexShader = compile vs_3_0 VS ();
                PixelShader = compile ps_3_0 PS ();
}
}
```

e. Extravasat render

```
// Declaring arrays
float4x4 WorldVP: WorldViewProjection;     // Work World, species and projection matrices
float4x4 mWorld: World;                    // world transformation matrix
float4x4 mWorldIT: WorldInverseTranspose;  // Inverted - transpose of the world matrix
float4x4 ViewInv: ViewInverse;             // Invert screen matrix


float sizeRatio: CHANNELVALUE0;
float acosCamAndRefl: CHANNELVALUE1;

float3 topScreenPos: CHANNELVECTOR0;
float3 botScreenPos: CHANNELVECTOR1;
float3 edgeScreenPos: CHANNELVECTOR2;
```

```
float4x4 matRot: CHANNELMATRIX0;

// samplers for texture
texture spriteMap: TEXTURE0;
sampler2D spriteSampler = sampler_state
{
Texture = <spriteMap>;
    MinFilter = Anisotropic;
    MagFilter = Anisotropic;
    MipFilter = Anisotropic;

// AddressU = WRAP;
// AddressV = WRAP;

    AddressU = BORDER;
AddressV =BORDER;

};


texture srcMap: TEXTURE1;
sampler2D srcSampler = sampler_state
{
Texture = <srcMap>;
    MinFilter = Anisotropic;
    MagFilter = Anisotropic;
    MipFilter = Anisotropic;

    AddressU = WRAP;
    AddressV = WRAP;

};

texture prevFrameMap: TEXTURE2;
sampler2D prevFrameSampler = sampler_state
{
Texture = <prevFrameMap>;
    MinFilter = Anisotropic;
    MagFilter = Anisotropic;
    MipFilter = Anisotropic;

};

texture obrezkaBluredMap: TEXTURE3;
sampler2D obrezkaBluredSampler = sampler_state
```

```
{
Texture = <obrezkaBluredMap>;
    MinFilter = Anisotropic;
    MagFilter = Anisotropic;
    MipFilter = Anisotropic;
};

texture frontSpriteMap: TEXTURE4;
sampler2D frontSpriteSampler = sampler_state
{
Texture = <frontSpriteMap>;
    MinFilter = Anisotropic;
    MagFilter = Anisotropic;
    MipFilter = Anisotropic;
};

texture obrezkaMap: TEXTURE5;
sampler2D obrezkaSampler = sampler_state
{
Texture = <obrezkaMap>;
    MinFilter = Anisotropic;
    MagFilter = Anisotropic;
    MipFilter = Anisotropic;
};



float2 ScreenToUV (float2 suv)
{
suv.y = -suv.y;
return (Suv +1) *0.5;
}

// IO structures
struct appdata {
    float4 Position: POSITION;
    float2 UV: TEXCOORD0;
};

struct vertexOutput {
float4 HPosition: POSITION;
float2  UV: TEXCOORD0;

};
```

```
vertexOutput VS (appdata IN)
{
vertexOutput OUT;
OUT.HPosition = IN.Position;
OUT.UV = IN.UV;


float4 p = IN.Position;
float4 wPos = mul (p, mWorld); // Get the vertex position in world space

return OUT;
}

float4 PS (vertexOutput IN): COLOR
{
float2 topSUV = ScreenToUV (topScreenPos.xy);
float2 botSUV = ScreenToUV (botScreenPos.xy);

float lenSUV = length (topSUV - botSUV);
float2 offset = {0.55.0.15};

float2 suv = (IN.UV - botSUV);

suv / = sizeRatio;
suv = mul (suv, matRot);
    suv * = 1/ LenSUV;
suv + = offset;

float4 spritePixel = tex2D (spriteSampler, suv);
float4 srcPixel = tex2D (srcSampler, botSUV);
float4 prevFramePixel = tex2D (prevFrameSampler, IN.UV);
float4    obrezkabluredPixel    =    tex2D    (obrezkaBluredSampler,    IN.UV-
float2(0.025.0.015)); // offset after smoothing
float4 obrezkaPixel = tex2D (obrezkaSampler, IN.UV);

float * (BotSUV) - lobrezka = saturate (length (IN.UV1/ (0.3* LenSUV)));
obrezkabluredPixel = lerp (obrezkaPixel, obrezkabluredPixel, lobrezka);

/// front sprite
float frontLratio = saturate ((4) * AcosCamAndRefl - 3);

float2 edgeSUV = ScreenToUV (edgeScreenPos.xy);
float frontLen = length (topSUV - edgeSUV);
suv = (IN.UV - topSUV);
    suv * = 1 / (5.5 * FrontLen);
```

```
        float2 frOffset = {0.5.0.5};
        suv + = frOffset;
        float4 frontSpritePixel = tex2D (frontSpriteSampler, suv);
        // float frontLratio = saturate (- (1 / lthresh) * frontLen + 1);
        // spritePixel = lerp (spritePixel, 1.1 * frontSpritePixel, frontLratio);
            spritePixel + = 1.1 * FrontSpritePixel * frontLratio;
        ///

        float lratio = saturate (obrezkabluredPixel.r);

        float br = srcPixel.r;
        const float stepuv = 0.01;
        float curstep = 0.01;
        for(int i =0; i <4; i ++)
        {
                br = max (tex2D (srcSampler, float2(BotSUV.x - curstep, botSUV.y)) r,
br);.
                br = max (tex2D (srcSampler, float2(BotSUV.x + curstep, botSUV.y)) r,
br).;
                br = max (tex2D (srcSampler, float2(BotSUV.x, botSUV.y - curstep)) r,
br);.
                br = max (tex2D (srcSampler, float2(BotSUV.x, botSUV.y + curstep)) r,
br).;
        curstep + = stepuv;
        }

        float3 res = spritePixel.rgb * br;
        res = max (res, prevFramePixel);

        res.rgb = lerp (float3(0.0.0), Res.rgb * 2, Lratio);

        return float4(Res,1);
        }

        technique HBlur
        {
        pass p0
        {
        CullMode =none;
                        AlphaBlendEnable = false;
                        VertexShader = compile vs_3_0 VS ();
                        PixelShader = compile ps_3_0 PS ();
        }
        }
```

f. Final contrast Render

```
/////////////////////////////////////////////
//////// Post Porcess Pic on Screen ///////////
/////////////////////////////////////////////
////////// Spiridonov Nikolay ////////////////
/////////////////////////////////////////////

// Declaring arrays
float4x4 WorldVP: WorldViewProjection;     // Work World, species and projection matrices
float4x4 mWorld: World;                    // world transformation matrix
float4x4 mWorldIT: WorldInverseTranspose;  // Inverted - transpose of the world matrix
float4x4 ViewInv: ViewInverse;             // Invert screen matrix


// samplers for texture
texture sourceTextureMap: TEXTURE0;
sampler2D sourceMapSampler = sampler_state
{
Texture = <sourceTextureMap>;
    MinFilter = Anisotropic;
    MagFilter = Anisotropic;
    MipFilter = Anisotropic;
};
texture capilarMap: TEXTURE1;
sampler2D capilarSampler = sampler_state
{
Texture = <capilarMap>;
// anntialiazing
    MinFilter = Anisotropic;
    MipFilter = Anisotropic;
    MagFilter = Anisotropic;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};

texture injectSummaryMap: TEXTURE2;
sampler2D injectSummarySampler = sampler_state
{
Texture = <injectSummaryMap>;
// anntialiazing
    MinFilter = Anisotropic;
    MipFilter = Anisotropic;
    MagFilter = Anisotropic;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};

texture refluxMap: TEXTURE3;
sampler2D refluxSampler = sampler_state
{
Texture = <refluxMap>;
// anntialiazing
    MinFilter = Anisotropic;
    MipFilter = Anisotropic;
    MagFilter = Anisotropic;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};


texture extravasatMap: TEXTURE4;
sampler2D extravasatSampler = sampler_state
{
Texture = <extravasatMap>;
// anntialiazing
    MinFilter = Anisotropic;
    MipFilter = Anisotropic;
    MagFilter = Anisotropic;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};

texture DistanceMapFromInject: TEXTURE5;
sampler2D DistanceMapFromInjectSampler = sampler_state
{
Texture = <DistanceMapFromInject>;
// anntialiazing
    MinFilter = Anisotropic;
    MipFilter = Anisotropic;
    MagFilter = Anisotropic;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};


float ratio (float4 pixel, float4 mask)
{
float res;
res = pixel.r *2;

return res;


};


// IO structures
struct appdata {
    float4 Position: POSITION;
    float2 UV: TEXCOORD0;
};

struct vertexOutput {
float4 HPosition: POSITION;
float2  UV: TEXCOORD0;

};

vertexOutput VS (appdata IN)
{
vertexOutput OUT;
OUT.HPosition = IN.Position;
OUT.UV = IN.UV;


float4 p = IN.Position;
float4 wPos = mul (p, mWorld); // Get the vertex position in world space

return OUT;
}

float4 PS (vertexOutput IN): COLOR
{

float4 inputPixel = tex2D (sourceMapSampler, IN.UV);
float4 capilarPixel = tex2D (capilarSampler, IN.UV);
```

```
float4 injectSummaryPixel = tex2D (injectSummarySampler, IN.UV);
float4 refluxPixel = tex2D (refluxSampler, IN.UV);
float4 extravasatPixel = tex2D (extravasatSampler, IN.UV);
float4 DistanceMapFromInjectPixel = tex2D (DistanceMapFromInjectSampler, IN.UV);


float4 resultPixel = {0.0.0.1};
float c =0.1;

extravasatPixel = max (extravasatPixel, refluxPixel);
float ratio = max (capilarPixel.r, injectSummaryPixel.r);
ratio = max (ratio, extravasatPixel.r);

if(Ratio < 0.5)
resultPixel.rgb = lerp (inputPixel.rgb,float3(C, c, c), ratio *2);
else
        resultPixel.rgb = float3(C, c, c) * ratio * 1.5;


return resultPixel;
}

technique HBlur
{
pass p0
{
CullMode =none;
                AlphaBlendEnable = false;
                VertexShader = compile vs_3_0 VS ();
                PixelShader = compile ps_3_0 PS ();
}}
```

g. polygon processing procedure for the blood texture

```
void BpGravity :: ProcessPolygon (void * Obj_data_ptr, Triangle tri, D3DXVECTOR3 g, int sign_cw_ccw,
Point <2> & G_pos, Point <2> & G_dir, D3DXVECTOR3 & g_global)
{

g_pos = invalid_pos;
g_dir = default_dir;
g_global = default_global;

int i0, i1, i2;
i0 = tri.i0;
i1 = tri.i1;
i2 = tri.i2;
```

```
        float w0, w1, w2;
            w1 = w2 = 0.33;
            w0 = 1 - w1 - w2;

        D3DXVECTOR3 p0 = ((Aco_DX8_ObjectDataChannel *) obj_data_ptr) -> GetVertexPosition (i0);
        D3DXVECTOR3 p1 = ((Aco_DX8_ObjectDataChannel *) obj_data_ptr) -> GetVertexPosition (i1);
        D3DXVECTOR3 p2 = ((Aco_DX8_ObjectDataChannel *) obj_data_ptr) -> GetVertexPosition (i2);

        D3DXVECTOR3 n0 = ((Aco_DX8_ObjectDataChannel *) obj_data_ptr) -> GetVertexNormals (i0);
        D3DXVECTOR3 n1 = ((Aco_DX8_ObjectDataChannel *) obj_data_ptr) -> GetVertexNormals (i1);
        D3DXVECTOR3 n2 = ((Aco_DX8_ObjectDataChannel *) obj_data_ptr) -> GetVertexNormals (i2);

        g_global = p0;
        if (Degenerate (p0, p1, p2))
        return;

        D3DXVECTOR3 p = p0 * w0 + p1 * w1 + p2 * w2;
        D3DXVECTOR3 n = CrossProduct (p1 - p0, p2 - p0);

        // if (DotProduct (n, n0) <0 || DotProduct (n, n1) <0 || DotProduct (n, n2) <0)
        // bool t = true;
        // n = n0 + n1 + n2;

        n / = Norm (n);
        if (Isnan (nx))
        return;

        D3DXVECTOR3 t = CrossProduct (n, CrossProduct (g, n));
        t / = Norm (t);
        if (Isnan (tx))
        return;

        float l0 = LengthSquared (p - (p0 + p1) / 2);
        float l1 = LengthSquared (p - (p1 + p2) / 2);
        float l2 = LengthSquared (p - (p2 + p0) / 2);
        float l = min (l0, min (l1, l2));
            l = sqrt (l) / 2;
        t * = l;

        D3DXVECTOR2 uv0 = ((Aco_DX8_ObjectDataChannel *) obj_data_ptr) -> GetVertexTUV (0, I0);
        D3DXVECTOR2 uv1 = ((Aco_DX8_ObjectDataChannel *) obj_data_ptr) -> GetVertexTUV (0, I1);
        D3DXVECTOR2 uv2 = ((Aco_DX8_ObjectDataChannel *) obj_data_ptr) -> GetVertexTUV (0, I2);
        if (Degenerate (D3DXVECTOR3 (uv0.x, uv0.y, 0), D3DXVECTOR3 (uv1.x, uv1.y, 0), D3DXVECTOR3 (uv2.x, uv2.y,
0)))
        return;
```

```
D3DXVECTOR2 e1 = uv0 - uv1;
D3DXVECTOR2 e2 = uv2 - uv1;
    D3DXVECTOR3 e1_3d (e1.x, e1.y, 0);
    D3DXVECTOR3 e2_3d (e2.x, e2.y, 0);
// exclude polys on seam
// if (sign_cw_ccw * DotProduct (CrossProduct (e2_3d, e1_3d), D3DXVECTOR3 (0, 0, 1)) <0)
// return;

D3DXVECTOR2 pos = uv0 * w0 + uv1 * w1 + uv2 * w2;
Barycentric (p0, p1, p2, p + t, w0, w1, w2);
D3DXVECTOR2 dir = uv0 * w0 + uv1 * w1 + uv2 * w2 - pos;
dir / = Norm (dir);

if (Isnan (dir.x))
return;

g_pos = Point <2> (Pos.x, pos.y);
g_dir = Point <2> (Dir.x, dir.y);

}
```

h. Monopolar rendering by sources

```
// Application matrix inputs
float4x4 object_to_clip: WORLDVIEWPROJECTION;

float pix_x: CHANNELVALUE0;
float pix_y: CHANNELVALUE1;
float power: CHANNELVALUE2;
float time: CHANNELVALUE3;
float sigma_sqr: CHANNELVALUE4;
float gauss_factor: CHANNELVALUE5;
float count: CHANNELVALUE6;
float maxDissectionEnergy: CHANNELVALUE7;
float etalonU: CHANNELVALUE8;
float etalonV: CHANNELVALUE9;
float UseEtalon: CHANNELVALUE10;
float DoUZV: CHANNELVALUE11;
// float UZVratio: CHANNELVALUE8; // RAS does not put in the texture of greater than this value if == - 1
is not stinging RAS

float2 src_0: CHANNELVECTOR0;
float2 src_1: CHANNELVECTOR1;
float2 src_2: CHANNELVECTOR2;
```

```
float2 src_3: CHANNELVECTOR3;
float2 src_4: CHANNELVECTOR4;
float2 src_5: CHANNELVECTOR5;
float2 src_6: CHANNELVECTOR6;
float2 src_7: CHANNELVECTOR7;
float2 src_8: CHANNELVECTOR8;
float2 src_9: CHANNELVECTOR9;
float2 src_10: CHANNELVECTOR10;
float2 src_11: CHANNELVECTOR11;
float2 src_12: CHANNELVECTOR12;
float2 src_13: CHANNELVECTOR13;
float2 src_14: CHANNELVECTOR14;
float2 src_15: CHANNELVECTOR15;f
float2 src_16: CHANNELVECTOR16;
float2 src_17: CHANNELVECTOR17;
float2 src_18: CHANNELVECTOR18;
float2 src_19: CHANNELVECTOR19;
float2 src_20: CHANNELVECTOR20;

// float2 etalonUV: CHANNELVECTOR20;

static const int max_src = 21;
static float2 src_ptr [max_src] =
{
src_0,
src_1,
src_2,
src_3,
src_4,
src_5,
src_6,
src_7,
src_8,
src_9,
src_10,
src_11,
src_12,
src_13,
src_14,
src_15,
src_16,
src_17,
src_18,
src_19,
src_20,
        };

        static float pix_area = pix_x * pix_y;
        static float2 pixel_size = float2(Pix_x, pix_y);
        static float2 uv_offset = 0.5 + 0.5 * Pixel_size;
        static  float4x4  uv_matrix  =  float4x4(0.5.  0.0.  0.0.  0.0.  0.0-0.5.  0.0.  0.0.  0.0.  0.0.  1.0.  0.0,
Uv_offset.x, uv_offset.y, 0.0. 1.0);

        // Textures and texture samplers
        texture alpha_texture: TEXTURE0;
        sampler2D alpha_sampler = sampler_state
        {
           Texture = <alpha_texture>;
           MinFilter = none;
           MagFilter = none;
           MipFilter = none;
           AddressU = clamp;
           AddressV = clamp;
        };

        // Vertex shader input structure
        struct in_vs
        {
        float4 pos: POSITION;
        float2uv0: TEXCOORD0;
        };

        // Pixel shader input structure
        struct in_ps
        {
        float4  clip_pos: POSITION;
        float4 uv_project: TEXCOORD0;
        };

        // Vertex shader
        in_ps main_vs (in_vs input)
        {
        in_ps output;
        output.clip_pos = mul (input.pos, object_to_clip);
        output.uv_project = mul (output.clip_pos, uv_matrix);
        return output;
        }

        float4 main_ps (in_ps input): COLOR
        {
        float2coord = input.uv_project.xy;
```

```
float3 old = (float3) Tex2D (alpha_sampler, coord);
float res = old.r;
old.g =0;
old.b =0;
bool dissection;
if (Power <0)
{
dissection =true;
power = -power;
}
else
dissection =false;


float max_inc = power * time * gauss_factor;
float inc = 0;

for (int i = 0; i <min (count, max_src); ++ i)
{
float2 src = src_ptr [i];
float dist = distance (coord, src.xy);
float p = gauss_factor * exp (- (dist * dist) / (2 * Sigma_sqr)); // * pix_area;
inc + = (power * time) * p;
}
float2 etalonUV = {etalonU, etalonV};
float distCheck =0.025f;//0.016f;
// if (etalonU == - 1)
// distCheck = 10;
if((float) Abs (etalonUV-coord) <distCheck ||! UseEtalon)
if(Dissection)
{
if(Res <=0.02)
{

? // inc = (inc> max_inc) inc: max_inc;
// res - = (abs (res-inc) <maxDissectionEnergy) inc: 0;
if (Inc> max_inc)
{
inc = max_inc;
}
if((Abs (res) + inc) <maxDissectionEnergy)
res- = inc;
}
}
else
{
```

```
if (DoUZV)
{
if(Res <=0.9)
{
if (Inc> max_inc)
{
inc = max_inc;
}
if((Abs (res) + inc) <maxDissectionEnergy)
res + = inc;
}
}
else
if(Res> =0)
{

if (Inc> max_inc)
{
res + = (res>0.5)? max_inc /2: Max_inc;
}
else
{
res + = (res>0.5)? Inc /2: Inc;
}
}
else
res * = -1;
}


return float4(Res, res, res, res);
}

// Techniques
technique Main
{
pass P0
{
        ZEnable = false;
        ZWriteEnable = false;
        VertexShader = compile vs_3_0 main_vs ();
        PixelShader = compile ps_3_0 main_ps ();
}
}
```

j. Generate subMesh

```
        void MiscTools :: MeshUtils :: GenerateSubMesh (Aco_Mesh *out, Aco_Mesh *in, Std ::vector<Size_t> &
vertices)//, D3DXVECTOR2 etUV)
        {
        MeshGraph mg;
            ProcessMesh (mg, in);
        MeshGraph :: TriListBuf & vertTriBuf = (* mg.vert_tri_ptr);
        MeshGraph :: TriBuf & triBuf = (* mg.tri_ptr);

        std :: set <size_t> aug_verts;
        std :: set <size_t> aug_polys;
        for (Std ::vector<Size_t> :: iterator it_v = vertices.begin (); ! It_v = vertices.end (); ++ it_v)
        {
        size_t iv = * it_v;

        for (MeshGraph :: TriList :: iterator it_tri = vertTriBuf [iv] .begin (); it_tri = vertTriBuf [iv] .end
();! ++ it_tri)
        {
        size_t itr = * it_tri;
        Triangle tr = triBuf [itr];

        if (Perimeter (in-> GetVertexTUV (0, Tr.i0), in-> GetVertexTUV (0, Tr.i1), in-> GetVertexTUV (0, Tr.i2))
<  0.15)
        {
        aug_polys.insert (itr);
        aug_verts.insert (tr.i0);
        aug_verts.insert (tr.i1);
        aug_verts.insert (tr.i2);
        }
        }
        }

        std :: map <size_t, size_t> perm;
        std ::vector<D3DXVECTOR3> res_verts;
        std ::vector<Triangle> res_polys;
        std ::vector<D3DXVECTOR3> res_norms;
        std ::vector<D3DXVECTOR2> res_tuv;

            size_t icur = 0;
        for (Std :: set <size_t> :: iterator it = aug_verts.begin (); it = aug_verts.end ();! ++ it)
        {
        size_t iv = * it;
        perm.insert (std :: pair <size_t, size_t> (iv, icur));
        ++ icur;
        res_verts.push_back (in-> GetVertexPosition (iv));
        res_norms.push_back (in-> GetVertexNormals (iv));
        // if (length (in-> GetVertexTUV (0, iv) - etUV) <0.005f)
                D3DXVECTOR2 ttuv = in-> GetVertexTUV (0, Iv);
        if (NearDots (ttuv, in-> GetVertexTUV (0, Vertices [0])))
        res_tuv.push_back (ttuv);
        else
        res_tuv.push_back (in-> GetVertexTUV (0, Vertices [0]));
        // else
        // res_tuv.push_back (etUV);
        // else
        // res_tuv.push_back (D3DXVECTOR2 (0.089021, 0.543895));

        }

        for (Std :: set <size_t> :: iterator it = aug_polys.begin (); it = aug_polys.end ();! ++ it)
        {
        size_t itr = * it;
        Triangle & tri = triBuf [itr];
        Triangle res;
        std :: map <size_t, size_t> :: iterator it_perm;

        it_perm = perm.find (tri.i0);
        res.i0 = it_perm-> second;
        it_perm = perm.find (tri.i1);
        res.i1 = it_perm-> second;
        it_perm = perm.find (tri.i2);
        res.i2 = it_perm-> second;

        res_polys.push_back (res);
        }

        GenerateMesh (out, Res_verts, res_polys, res_norms, res_tuv);

        }


        void GenerateMesh (Aco_Mesh * ch, std ::vector<D3DXVECTOR3> & verts, std ::vector<Triangle> & polys, std
::vector<D3DXVECTOR3> & norms, std ::vector<D3DXVECTOR2> & tuv)
        {
        size_t vcnt = verts.size ();
        size_t tcnt = polys.size ();

        ch-> Release ();
        ch-> SetVertexCount (vcnt);
            ch-> SetIndexCount (tcnt * 3);
```

```
for (Size_t iv = 0; iv <vcnt; ++ iv)
{
ch-> SetVertexPosition (verts [iv], iv);
ch-> SetVertexNormal (norms [iv], iv);
ch-> SetVertexTUCoord (0, Tuv [iv], iv);

}

for (Size_t it = 0; it <tcnt; ++ it)
{
Triangle tri = polys [it];

        ch-> SetIndex (tri.i0, it * 3);
        ch-> SetIndex (tri.i1, it * 3 + 1);
        ch-> SetIndex (tri.i2, it * 3 + 2);
}

ch-> CreateVertexBuffer ();
}
```

k. Submesh Rendering shader

```
// Declaring arrays
float4x4 object_to_clip: WorldViewProjection;      // Work World, species and projection matrices
float4x4 object_to_world: World;                   // world transformation matrix
float4x4 world_to_object: WorldInverseTranspose;   // Inverted - transpose of the world matrix
float4x4 view_to_world: ViewInverse;               // Invert screen matrix

float3 box_size: CHANNELVECTOR0;
// float3 etalonUV: CHANNELVECTOR1;
float4x4 box_pose: CHANNELMATRIX0;

// Vertex shader input structure
struct in_vs
{
float4 pos: POSITION;
float2 uv0: TEXCOORD0;
};

// Pixel shader input structure
struct in_ps
{
float4 pos: POSITION;
float2 uv0: TEXCOORD0;
float3 wpos: TEXCOORD1;
};

float3 Vec3TransformCoord (float3 v, float4x4 m)
{
float x, y, z, w;
x = vx * m [0] [0] + Vy * m [1] [0] + Vz * m [2] [0] + M [3] [0];
y = vx * m [0] [1] + Vy * m [1] [1] + Vz * m [2] [1] + M [3] [1];
z = vx * m [0] [2] + Vy * m [1] [2] + Vz * m [2] [2] + M [3] [2];
w = vx * m [0] [3] + Vy * m [1] [3] + Vz * m [2] [3] + M [3] [3];

return float3(X / w, y / w, z / w);
}

float IsInsideSphere (float3 p, float3 p0, float r)
{
float res = 0;
if (Length (p - p0) <r)
        res = 1;

return res;
}

float IsInsideCylinder (float3 testpt, float3 pt1, float3 pt2, float radius_sq)
{
float lengthsq = pow (length (pt1 - pt2), 2.0);
float dx, dy, dz;   // vector d from line segment point 1 to point 2
float pdx, pdy, pdz;    // vector pd from point 1 to test point
float dot, dsq;

dx = pt2.x - pt1.x;// translate so pt1 is origin. Make vector from
    dy = pt2.y - pt1.y;     // pt1 to pt2. Need for this is easily eliminated
dz = pt2.z - pt1.z;

pdx = testpt.x - pt1.x;// vector from pt1 to test point.
pdy = testpt.y - pt1.y;
pdz = testpt.z - pt1.z;

// Dot the d and pd vectors to see if point lies behind the
// cylinder cap at pt1.x, pt1.y, pt1.z

dot = pdx * dx + pdy * dy + pdz * dz;

// If dot is less than zero the point is behind the pt1 cap.
// If greater than the cylinder axis line segment length squared
// then the point is outside the other end cap at pt2.

if(Dot < 0.0f || dot> lengthsq)
{
return(-1.0f);
}
else
{
// Point lies within the parallel caps, so find
// distance squared from point to line, using the fact that sin ^ 2 + cos ^ 2 = 1
// the dot = cos () * | d || pd |, and cross * cross = sin ^ 2 * | d | ^ 2 * | pd | ^ 2
// Carefull: '*' means mult for scalars and dotproduct for vectors
// In short, where dist is pt distance to cyl axis:
// dist = sin (pd to d) * | pd |
// distsq = dsq = (1 - cos ^ 2 (pd to d)) * | pd | ^ 2
// dsq = (1 - (pd * d) ^ 2 / (| pd | ^ 2 * | d | ^ 2)) * | pd | ^ 2
// dsq = pd * pd - dot * dot / lengthsq
// where lengthsq is d * d or | d | ^ 2 that is passed into this function

// distance squared to the cylinder axis:

dsq = (pdx * pdx + pdy * pdy + pdz * pdz) - dot * dot / lengthsq;

if(Dsq> radius_sq)
{
return(-1.0f);
}
else
{
return(Dsq);// return distance squared to axis
}
}
}
```

```
float IsInsideCapsule (float3 p, float4x4 box_pose, float3 box_size)
{
float3 pt1 = {-0.5.0.0}, Pt2 = {0.5.0.0}, Pt0, axis;
float bx, by, bz, r;

bx = box_size.x;
by = box_size.y;
bz = box_size.z;
r = bz;

pt1 * = box_size;
pt2 * = box_size;

pt1 = Vec3TransformCoord (pt1, box_pose);
pt2 = Vec3TransformCoord (pt2, box_pose);
// axis = normalize (pt2 - pt1);
// pt0 = (pt1 + pt2) / 2.0;
// pt1 = pt0 - axis * bx / 2;
// pt2 = pt0 + axis * bx / 2;

float res = 0;
if (IsInsideCylinder (p, pt1, pt2, r * r)> = 0)
        res = 1;
else if (IsInsideSphere (p, pt1, r))
        res = 1;
else if (IsInsideSphere (p, pt2, r))
        res = 1;

return res;
}

float IsInsideBox (float3 p, float4x4 box_pose, float3 box_size)
{
float3 box_orig;
float3 x_axis;
float3 y_axis;
float3 z_axis;
float3 tmp_vec;

float dx = box_size.x * 2;
float dy = 0;
float dz = 0;
float3 trans = {dx, dy, dz};
tmp_vec = Vec3TransformCoord (trans, box_pose) - Vec3TransformCoord (float3(0.0.0), Box_pose);
box_pose [3] [0] + = Tmp_vec.x;
box_pose [3] [1] + = Tmp_vec.y;
box_pose [3] [2] + = Tmp_vec.z;

    box_size.x = (box_size.x / 2.0 + Length (tmp_vec)) * 2.0;

// float bscale = 1.4;
//box_size.x * = bscale;
//box_size.y * = bscale;
//box_size.z * = bscale;

box_orig = Vec3TransformCoord (float3(0.0.0), Box_pose);

tmp_vec = Vec3TransformCoord (float3(1.0.0), Box_pose);
x_axis = normalize (tmp_vec - box_orig);
tmp_vec = Vec3TransformCoord (float3(0.1.0), Box_pose);
y_axis = normalize (tmp_vec - box_orig);
tmp_vec = Vec3TransformCoord (float3(0.0.1), Box_pose);
z_axis = normalize (tmp_vec - box_orig);

float3 pos = p - box_orig;

float px = dot (pos, x_axis);
float py = dot (pos, y_axis);
float pz = dot (pos, z_axis);

float bx = box_size.x / 2.0;
float by = box_size.y / 2.0;
float bz = box_size.z / 2.0;

float res = 0;
if (Abs (px) <bx && abs (py) <by && abs (pz) <bz)
        res = 1;

return res;
}

// Vertex shader
in_ps main_vs (in_vs input)
{
in_ps output;

    output.pos.xy = input.uv0.xy * 2.0 - 1.0;
output.pos.y * = -1;
    output.pos.z = 0;
    output.pos.w = 1;
output.uv0 = input.uv0;
output.wpos = mul (input.pos, object_to_world);

return output;
}

float4 main_ps (in_ps input): COLOR
{
float res = 0;
if (IsInsideCapsule (input.wpos, box_pose, box_size))
// res = abs (etalonUV-input.uv0) <0.02 1: 0;
res =1;
else
clip (-1);

return float4(Res, res, res, res);
}

// Techniques
technique Main
{
pass P0
{
        ZEnable = false;
        ZWriteEnable = false;
        CullMode = none;
        VertexShader = compile vs_3_0 main_vs ();
        PixelShader = compile ps_3_0 main_ps ();
}
}

technique Main_cw
{
pass P0
{
        ZEnable = false;
        ZWriteEnable = false;
CullMode = cw;
        VertexShader = compile vs_3_0 main_vs ();
        PixelShader = compile ps_3_0 main_ps ();
}
}

technique Main_ccw
{
pass P0
{
        ZEnable = false;
        ZWriteEnable = false;
CullMode = ccw;
        VertexShader = compile vs_3_0 main_vs ();
        PixelShader = compile ps_3_0 main_ps ();
}
}
```

m. Organ shader

```
// DIFFUSEMAP + SPECULARMAP + NORMALMAP + LIGHTMAP
////////////////////////////////////////////////// //////////////////////
///// variable declaration ////////////////////////////////////////// //
////////////////////////////////////////////////// //////////////////////
// Declaring arrays
float4x4 WorldVP: WorldViewProjection;     // Work World, species and projection matrices
float4x4 mWorld: World;                    // world transformation matrix
float4x4 mWorldIT: WorldInverseTranspose;  // Inverted - transpose of the world matrix
float4x4 ViewInv: ViewInverse;             // Invert screen matrix

// macro to replace the code in MAX_LIGHTS 8
#define MAX_LIGHTS 8

// Light source
float3 lightpos [MAX_LIGHTS]: LIGHTPOS;     // Connect light source position
float3 lightcol [MAX_LIGHTS]: LIGHTCOLOR;   // Connect source color

float distToGlow: CHANNELVALUE0;
float c2: CHANNELVALUE1;
float maxHemRadius: CHANNELVALUE2;
float c4: CHANNELVALUE3;
float mdist: CHANNELVALUE4; // max dist from vertex
float hemCount: CHANNELVALUE5;
float burnLerp: CHANNELVALUE6;
float normalRevert: CHANNELVALUE7;
float cutContourUVSet: CHANNELVALUE8;
float vhintTm: CHANNELVALUE9;
float hemMapUse: CHANNELVALUE10;
float uKa: CHANNELVALUE11;
float uKd: CHANNELVALUE12;
float uKs: CHANNELVALUE13;
float bumpScale: CHANNELVALUE14;
float shininess: CHANNELVALUE15;
float falloff: CHANNELVALUE16;
float angualarFalloff: CHANNELVALUE17;
float dofClampNear: CHANNELVALUE18;
float dofClampFar: CHANNELVALUE19;
float diffuseUVSet: CHANNELVALUE20;
float Blend: CHANNELVALUE21;                  // DOF or Blending

float3 AmbientColor: CHANNELVECTOR0;
float3 DiffuseColor: CHANNELVECTOR1;
float3 SpecularColor: CHANNELVECTOR2;
float3 posGem: CHANNELVECTOR3; // World position of creating a hematoma
float3 hemAddColor: CHANNELVECTOR4;
float3 bpKaKdKs: CHANNELVECTOR5;
float3 bpBuSh: CHANNELVECTOR6;
float3 burnSatBri: CHANNELVECTOR7;
float3 dofParams: CHANNELVECTOR8;
float3 coag_color: CHANNELVECTOR9;

float4x4 hemMatrix1: CHANNELMATRIX0;
float4x4 hemMatrix2: CHANNELMATRIX1;
```

```
// Textures
texture diffuseTextureMap: TEXTURE0;// Diffuse texture
sampler diffuseTextureSampler = sampler_state
{
Texture = <diffuseTextureMap>;
// antializing
    MinFilter = Anisotropic;
    MipFilter = Anisotropic;
    MagFilter = Anisotropic;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};

texture normalMap: TEXTURE1;// Texture Normal
sampler normalSampler = sampler_state
{
Texture = <normalMap>;
// antializing
MinFilter = Linear;
    MipFilter = None;
MagFilter = Linear;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};

texture bpBumpMap: TEXTURE2;// Bump blood
sampler bpBumpSampler = sampler_state
{
Texture = <bpBumpMap>;
// antializing
MinFilter = Linear;
    MipFilter = none;
MagFilter = Linear;

// addressing mode
    AddressU = clamp;
    AddressV = clamp;
};

texture bloodMap: TEXTURE3;// Map the blood
sampler bloodSampler = sampler_state
{
Texture = <bloodMap>;
// antializing
MinFilter = Linear;
MipFilter = Linear;
MagFilter = Linear;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};

texture burnMap: TEXTURE4;// Map burns
sampler burnSampler = sampler_state
{
Texture = <burnMap>;
// antializing
    MinFilter = Anisotropic;
    MipFilter = Anisotropic;
    MagFilter = Anisotropic;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};

texture cutDoubleMap: TEXTURE5;// incisions Map
sampler cutDoubleSampler = sampler_state
{
Texture = <cutDoubleMap>;
// antializing
    MinFilter = Anisotropic;
    MipFilter = Anisotropic;
    MagFilter = Anisotropic;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};

texture vhintMap: TEXTURE6;// tips Map
sampler vhintSampler = sampler_state
{
Texture = <vhintMap>;
// antializing
    MinFilter = Anisotropic;
    MipFilter = Anisotropic;
    MagFilter = Anisotropic;
// addressing mode
    AddressU = Wrap;
    AddressV = Wrap;
    AddressW = Wrap;
};





////////////////////////////////////////////////// ///////////////////////
///// Structure ////////////////////////////////////////////// ////////////
////////////////////////////////////////////////// ///////////////////////

// Incoming data
struct appData
{
float4 position: POSITION;          // Position the object
float3 Normal: NORMAL;            // The normal to the surface
float4 color0: COLOR0;            // vertex color
float2 texcoord0: TEXCOORD0;         // Texture coordinates ID = 1
float2 texcoord1: TEXCOORD1;         // Texture coordinates ID = 2
float2 texcoord2: TEXCOORD2;         // Texture coordinates ID = 3
};

// Outgoing data
struct vertexOutput
{
float4 HPosition: POSITION;          // Position the object
float4 color0: COLOR0;            // vertex color
float4 wPosition: TEXCOORD0;
float3 wNormal: TEXCOORD1;          // normal direction in space
float3 LightVector: TEXCOORD2;         // Vector light
float3 EyeVector: TEXCOORD3;         // Vector observer
float2 texcoord0: TEXCOORD4;         // Texture coordinates
float2 texcoord1: TEXCOORD5;         // Texture coordinates
float2 texcoord2: TEXCOORD6;         // Texture coordinates
float3 world_pos: TEXCOORD7;
};

float formula (float arg1) {
float arg = (arg1 <0)?0: ((Arg1>1)?1: Arg1);
```

```
// float gg = (arg> c4) arg: (pow (arg, 3) + (c4-pow (c4,3)));?
floata = (arg <c4)? pow (c4, -2): (C4-1) / (Pow (c4, -3) -1);
float b = (arg <c4)?0:1-a;
float s = (arg <c4)?3: -3;
// float gg = (arg> c4) arg: (pow (c4, -2) * pow (arg, 3));?
float gg = a * pow (arg, s) + b;
return gg;
}

float getHematomaIntensity (float2 tc) { // intensity in point tc
? // int hc = (hemCount <8) (int) hemCount: 8;
// float2 coord;
float dst;
float OUT0 = 0;
float out1 = 0;
int i = 0;
float4 mat;

while (I <hemCount) {
// mat = (i <4) (hemMatrix1 [i]) :( hemMatrix2 [i-4])?;
if (I <4) {
mat = hemMatrix1 [i];
}
else{
mat = hemMatrix2 [i-4];
}
// coord = mat.xy;
dst = distance (mat.xy, tc);
if (Dst <mat.z) {
out1 = (1-dst / mat.z);
if (Out1> out0) {
out0 = out1;
}
}
i ++;
}
return OUT0;
}

float2 dHdxy_derivmap (sampler sampler_in, float2 texST, bool bFlipVertDeriv = false)
{
float3 normal = tex2D (sampler_in, texST);
float3 dHdST = 2.0 * Normal - 1.0;
dHdST * = -1.0 / DHdST.z;

dHdST.y * = (bFlipVertDeriv? -1.0 : 1.0);
// chain rule
float2 dSTdx = ddx (texST);
float2 dSTdy = ddy (texST);
return float2(Dot (dHdST.xy, dSTdx.xy), dot (dHdST.xy, dSTdy.xy));
}

float3 perturbNormal (float3 surf_pos, float3 surf_norm, float2 dHdxy)
{
float3 vSigmaX = ddx (surf_pos);
float3 vSigmaY = ddy (surf_pos);
float3 vN = surf_norm; // normalized

float3 R1 = cross (vSigmaY, vN);
float3 R2 = cross (vN, vSigmaX);

float fDet = dot (vSigmaX, R1);

float3 vGrad = sign (fDet) * (dHdxy.x * R1 + dHdxy.y * R2);
return normalize (abs (fDet) * surf_norm - vGrad);
}

float3 getBurnColor (float3 diff, float sat, float bri)
{
        float gray = 0.3 * Diff.r + 0.59 * Diff.g + 0.11 * Diff.b;
        float3 burn;

        burn.r = lerp (diff.r, coag_color.r, sat);// lerp (diff.r, gray, sat);
        burn.g = lerp (diff.g, coag_color.g, sat);//lerp(diff.g, gray, sat);
        burn.b = lerp (diff.b, coag_color.b, sat);// lerp (diff.b, gray, sat);


        burn.rgb * = bri;

        return burn;
        }

        float2 getBurnLevel (float energy)
        {
        float alpha = 2.0 * (1.0 - exp (-log (2.0) * Energy));
        float beta = saturate (alpha - 0.85);

        alpha = saturate (alpha);
            alpha = 1.0 / (1.0 + Exp (-40.0 * (Alpha - 0.1)));

        return float2(Alpha, beta);
        }

        float ComputeDepthBlur (float fDepth, float3 vDofParams, float clampNear, float clampFar)
        {
        // vDofParams coefficients
        // x = near blur depth; y = focal plane depth; z = far blur depth;
        // clampNear = blurriness cutoff constant for objects in front of the focal plane;
        // clampFar = blurriness cutoff constant for objects behind the focal plane;

        float f;
        if (FDepth <vDofParams.y)
        {
        // scale depth value between near blur distance and focal distance
        // to [-1, 0] range
        f = (fDepth - vDofParams.y) / (vDofParams.y - vDofParams.x);
        f = -1.0 + 1.0 / (1.0 + Exp (-10.0 * (F + 0.4)));
        // clamp the near blur to a maximum blurriness
                f = clamp (f, -clampNear, 0);
        }
        else
        {
        // scale depth value between focal distance and far blur distance
        // to [0, 1] range
        f = (fDepth - vDofParams.y) / (vDofParams.z - vDofParams.y);
                f = 1.0 / (1.0 + Exp (-10.0 * (F - 0.4)));
        // clamp the far blur to a maximum blurriness
                f = clamp (f, 0, ClampFar);
        }
        // scale and bias into [0, 1] range
        return f * 0.5f + 0.5f;
        }

        ////////////////////////////////////////////////// //////////////////////
        ///// Shaders ////////////////////////////////////////////// //////////////
        ////////////////////////////////////////////////// //////////////////////

        // Vertex shader
        vertexOutput VS (appData IN)
        {
            vertexOutput OUT; // Reset output structure

            OUT.wNormal = normalRevert * normalize (mul (IN.Normal, mWorld)); // Get the normals in the world
coordinate system

        float4 p = IN.position;
        OUT.HPosition = mul (p, WorldVP);
        OUT.world_pos = OUT.HPosition;

        float4 wPos = mul (p, mWorld); // Get the vertex position in world space
        OUT.wPosition = wPos;
        OUT.LightVector = lightpos [0] - wPos; // vector of light in space
        OUT.EyeVector = ViewInv [3] - wPos; // Vector observer in space

            OUT.texcoord0 = IN.texcoord0; // pass the texture coordinates in the pixel shader
        OUT.texcoord1 = IN.texcoord1;
        OUT.texcoord2 = IN.texcoord2;

        float4 clr = {0.0f, 0.0f, 0.0f, 0.0};
        bool f = length (IN.color0) == 0;
        if(F)
                OUT.color0 = float4(1.0. 1.0. 1.0. 0);
        else
        OUT.color0 = IN.color0;

        return OUT;
        }

        // Pixel Shader
        float4 PS (vertexOutput IN): COLOR
        {
        float3 L = normalize (IN.LightVector); // Create a vector of the light
        float3 V = normalize (IN.EyeVector); // Create a vector of the observer

        float2 uv_diffuse = {0.0};
        if (DiffuseUVSet == 0)
        uv_diffuse = IN.texcoord0.xy;
        else if (DiffuseUVSet == 1)
        uv_diffuse = IN.texcoord1.xy;
        else if (DiffuseUVSet == 2)
        uv_diffuse = IN.texcoord2.xy;
        else
        uv_diffuse = IN.texcoord0.xy;
        float4 g = {0.1.0.0};

        float4 texDiff = tex2D (diffuseTextureSampler, uv_diffuse); // Read the diffuse texture
        float3 bloodTex = (float3) Tex2D (bloodSampler, IN.texcoord0.xy);
        float3 burnTex = (float3) Tex2D (burnSampler, IN.texcoord0.xy);
        float4 cutDoublePixel = tex2D (cutDoubleSampler, uv_diffuse);

             // alpha in pixel == 1 then use cut texture
        bool inCut = false;
        if(IN.color0.a> 0)
        {
                inCut = true;
        // texDiff = float4 (1.0, 1.0, 1.0, 1.0);
        texDiff = cutDoublePixel;
        }
            IN.color0.a = 1.0;
        texDiff * = IN.color0;

        float2 burn_level = getBurnLevel (abs (burnTex.r));
        float3 burn_color = getBurnColor (texDiff.rgb, burnSatBri.x, burnSatBri.y);


        if (BloodTex.r! = 1 && bloodTex.g! = 1 && bloodTex.b! = 1)
                bumpScale = bumpScale / 2.0;


        float3 N = normalize (IN.wNormal);
        if (BumpScale! = 0.0)
        {
        float2 grad = dHdxy_derivmap (normalSampler, uv_diffuse, true);
        grad * = bumpScale * (1.0 + 10.0 * Burn_level.x);
        N = perturbNormal (IN.wPosition, N, grad);
```

```
        grad = dHdxy_derivmap (bpBumpSampler, IN.texcoord0.xy, true);
grad * = bpBuSh.x;
N = perturbNormal (IN.wPosition, N, grad);
}



// mixing blood and coagulation
texDiff.rgb = lerp (texDiff.rgb, burn_color, burn_level.x * texDiff.a);
    texDiff.rgb = lerp (texDiff.rgb, float3(0.0.0), Burn_level.y * texDiff.a);
if(! InCut)
texDiff.xyz = lerp (texDiff.xyz, texDiff.xyz * bloodTex, texDiff.a);

// tips
if (VhintTm>0 &&! InCut)
{
float4 texVHint = tex2D (vhintSampler, IN.texcoord0.xy);

if(TexVHint.a ==1)
texDiff = lerp (texDiff, texVHint, vhintTm *0.6* (TexVHint.r + texVHint.g + texVHint.b));
else
texDiff = lerp (texDiff, texVHint,1-texVHint.a);

}

float diffuse = saturate (dot (L, N));
float dist = distance (lightpos [0], IN.wPosition);


float3 H = normalize (L + V); // Half - vector
float dotNH = saturate (dot (N, H));

float specular = pow (dotNH, shininess);
float k = diffuse * (shininess + 6.0) / 8.0 / 3.14;
specular * = k;
uKs * = max (0.05, Pow (1.0 - burn_level.x, 2.0));


float att = saturate (1.0 - saturate ((dist * dist) / (falloff * falloff)));
float3 viewVector = mul (float3(0. 0. 1), ViewInv);
att * = pow (dot (L, viewVector), angualarFalloff);
float att2 = att * att;

// Formation of the pixel color
float4 res = float4(UKa * att2 * texDiff.rgb, 1);
res.rgb + = uKd * att2 * saturate (texDiff * diffuse);
res.rgb + = uKs * att2 * saturate (SpecularColor * specular);
res.rgb + = saturate (dist <distToGlow?2* (Dist-distToGlow) * (dist-distToGlow):0);
float alpha = 1;
/ *
if (Blend == 1)
alpha = texDiff.a;
else
alpha = (cutTex.r <0.5)? pow (2 * cutTex.r, 8): 1;
* /
return float4(Res.rgb, alpha);
}

float4 PS_DOF (vertexOutput IN): COLOR
{
return float4(0. 0. 0, ComputeDepthBlur (IN.world_pos.z, dofParams, dofClampNear, dofClampFar));
}

///////////////////////////////////////////////// //////////////////////
///// Equipment /////////////////////////////////////////// //////////////
///////////////////////////////////////////////// //////////////////////
```

```
technique Main
{
pass p0
{
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS ();
        ZWriteEnable = true;
}
}

technique MainBackface
{
pass p0
{
CullMode = cw;
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS ();
        ZWriteEnable = true;
}
}

technique RGB_DOF
{
pass p0
{
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS ();
        ZWriteEnable = true;
}

pass p1
{
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS_DOF ();
        ZWriteEnable = true;
ColorWriteEnable = Alpha;
        AlphaBlendEnable = false;
}
}

technique RGB_DOF_Cw
{
pass p0
{
CullMode = cw;
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS ();
        ZWriteEnable = true;
}

pass p1
{
CullMode = cw;
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS_DOF ();
        ZWriteEnable = true;
ColorWriteEnable = Alpha;
        AlphaBlendEnable = false;
}
}

technique DOF_in_Alpha
{
pass p0
{
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS_DOF ();
        ZWriteEnable = true;
ColorWriteEnable = Alpha;
```

```
        AlphaBlendEnable = false;
}
}
technique DofForTransparentWithCW //5
{
pass p0
{
CullMode = cw;
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS ();
        ZWriteEnable = true;
AlphaBlendEnable =true;
}

pass p1
{
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS_DOF ();
        ZWriteEnable = true;
ColorWriteEnable = Alpha;
        AlphaBlendEnable = false;
}
}
technique DofForTransparentWithCCW // 6
{
pass p0
{
CullMode = ccw;
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS ();
        ZWriteEnable = true;
AlphaBlendEnable =true;
}

pass p1
{
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS_DOF ();
        ZWriteEnable = true;
ColorWriteEnable = Alpha;
        AlphaBlendEnable = false;
}
}
technique DofForTransparentWithNone // 7
{
pass p0
{
CullMode =none;
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS ();
        ZWriteEnable = true;
AlphaBlendEnable =true;
}

pass p1
{
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS_DOF ();
        ZWriteEnable = true;
ColorWriteEnable = Alpha;
        AlphaBlendEnable = false;
}
}
technique RGB_DOF_withNoneCulling
{
pass p0
{
CullMode =none;
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS ();
        ZWriteEnable = true;
}

pass p1
{
        VertexShader = compile vs_3_0 VS ();
        PixelShader = compile ps_3_0 PS_DOF ();
        ZWriteEnable = true;
ColorWriteEnable = Alpha;
        AlphaBlendEnable = false;
}
```