Prof. Dr.-Ing. Ralf Steinmetz
Multimedia communications Lab

Dr.-Ing. Florian Mehm
Dipl.-Inform. Robert Konrad

Game Technology
Winter Semester 2016/2017

Solution 2

General Information

- The exercises may be solved by teams of up to three people.
- The solutions have to be uploaded to the Git repositories assigned to the individual teams.
- **The submission date (for practical and theoretical tasks) is noted on top of each exercise sheet.**
- If you have questions about the exercises write a mail to game-technology@kom.tu-darmstadt.de or use the forum at https://www.fachschaft.informatik.tu-darmstadt.de/forum/viewforum.php?f=557

P2 Practical Task: Crack me up (5 Points)

Old games were often accompanied by little start animations called “cracktros”. Most cracktros are simply functional animations of positions and colors. YouTube contains hundreds of examples.

In this exercise, you can build upon the code you wrote for exercise 1 if you want to. Animate at least five properties of your graphics in a purely functional way. Example properties could be position, rotation, scale, colors, and many more.

Notes:

- If your texture(s) use PNG, drawTexture will use the alpha channel for alpha blending
- Unless you write your own code, there is no text rendering code in Kore

https://github.com/TUDGameTechnology/Exercise2.git contains additional code to help you out. You can either copy the code changes manually or just pull them into your own repository using
git pull https://github.com/TUDGameTechnology/Exercise2.git (and then handle any merges necessary).

Please remember to push into a branch called exercise2.

Thank you for the creative solutions to this task! There is no code solution for this exercise.

T2. Theoretical Tasks: Timing (5 Points)

T2.1 Iterative and functional time calculations (1 Point)

A space ship labeled Vic Viper starts at time t = 0 and position x = 10 with an x speed of 14 per second. Calculate the position x after 5 seconds (a) procedurally and (b) iteratively with 3 steps per second. (You can write a script or use a spreadsheet application).

a) When we calculate the position procedurally, we find a closed formula for the position depending on time:

$$f(t) = x_0 + v \cdot t$$
$$f(t) = 10 + 14 \cdot t$$
$$f(5) = 10 + 14 \cdot 5 = 80$$

b) For calculating iteratively, we calculate the updated position based on the time delta from the last time and the previous position.

$$f(t + \Delta t) = f(t) + v \cdot \Delta t$$

When we iteratively apply this formula, we get the solution f(5) = 80.

Time	DeltaT	Pos	Speed
0	0,33	10	14
0,33333333	0,33	14,6666667	14
0,66666667	0,33	19,3333333	14
1	0,33	24	14
1,33333333	0,33	28,6666667	14
1,66666667	0,33	33,3333333	14
2	0,33	38	14
2,33333333	0,33	42,6666667	14
2,66666667	0,33	47,3333333	14
3	0,33	52	14
3,33333333	0,33	56,6666667	14
3,66666667	0,33	61,3333333	14
4	0,33	66	14

4,33333333	0,33	70,6666667	14
4,66666667	0,33	75,3333333	14
5	0,33	80	14

In this case, we arrive at the same solution, since this is an easy calculation. In most cases in game development, it is very hard or infeasible to derive a closed solution, which is why we often have to apply the iterative method of calculation.

T2.2 Different framerates (1 Point)

The situation starts out as described in 2.1 but there is a wall of width 5 at between positions x1= 60 and x2=65 (the space ship is a point, it has no width). Calculate the position of the space ship after 5 seconds for (a) a 3 steps per second and (b) a 2 steps per second update. Check for collisions in every update step. When a collision happens, move back the ship just so far that it barely touches the wall and set its speed to 0.

a) We see a collision at time t = 3 2/3 s, therefore, we set the position back and reduce the speed to 0.

Time	Delta	Pos	Speed
0	0,33	10	14
0,33333333	0,33	14,6666667	14
0,66666667	0,33	19,3333333	14
1	0,33	24	14
1,33333333	0,33	28,6666667	14
1,66666667	0,33	33,3333333	14
2	0,33	38	14
2,33333333	0,33	42,6666667	14
2,66666667	0,33	47,3333333	14
3	0,33	52	14
3,33333333	0,33	56,6666667	14
3,66666667	0,33	61,3333333	14
4	0,33	60	0
4,33	0,33	60	0
4,66	0,33	60	0
4,99	0,33	60	0

b) The collision between times 3.5 and 4 is missed, we continue flying as if nothing happened.

Time	DeltaT	Pos	Speed
0	0,50	10	14
0,5	0,50	17	14
1	0,50	24	14
1,5	0,50	31	14

2	0,50	38	14
2,5	0,50	45	14
3	0,50	52	14
3,5	0,50	59	14
4	0,50	66	14
4,5	0,50	73	14
5	0,50	80	14

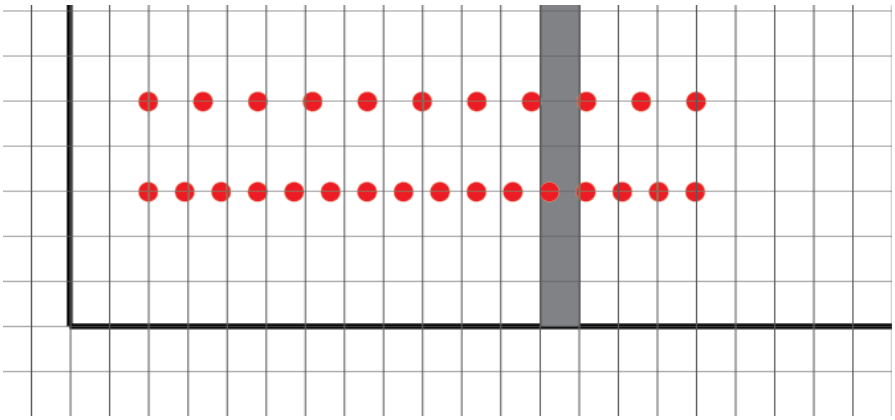


Figure 1: Upper row: Positions for deltaT = 1/2s, Lower row: Positions for deltaT = 1/3s

We see the effect of "tunneling", i.e. the space ship has travelled past the obstacle in one frame. This can happen if the simulated object was on one side of the obstacle in one frame and on the other side on the next frame. Without other checks, e.g. casting a ray between the two positions and checking if there is an obstacle that should have been hit, we will not be able to find this hit. This is the reason why especially fast-travelling objects such as bullets are often handled with special code in games.

T2.3 Framerate (0.5 Points)

You are working on a game without using any form of multitasking. As it turns out, a purely graphical effect you implemented runs slow. A colleague suggests calculating and caching the effect every third frame only. Is this a proper optimization strategy? Discuss the resulting changes to the framerate and the game experience.

Overall, the number of frames per time unit would increase, when we assume that calculating the effect once and caching it is faster than calculating it each frame.

The game experience might suffer from two factors:

- *The players might notice that the effect is cached, i.e. the perceived framerate of the effect is lower than the actual framerate.*
- *The frame rate is not steady: During the frame in which the effect is calculated, it is lower, and higher during the other frames. Especially in VR applications, this can lead to a jarring feeling.*

All in all, the goal should be to render each frame optimally, so the proper optimization should rather be focused on optimizing the effect itself.

T2.4 Vertical Synchronization (1.5 Points)

Consider a monitor that runs with 60 Hz and a game that runs vsynced and which consistently takes 10 milliseconds to render one frame. How much time does the game spend waiting for the monitor (vsync) on average when

a) double buffering,
b) triple buffering with the mode introduced as FIFO,
c) triple buffering with the mode introduced as MAILBOX

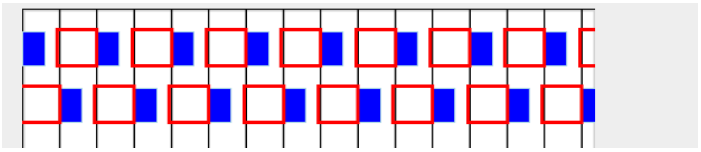
is used?

Notes:

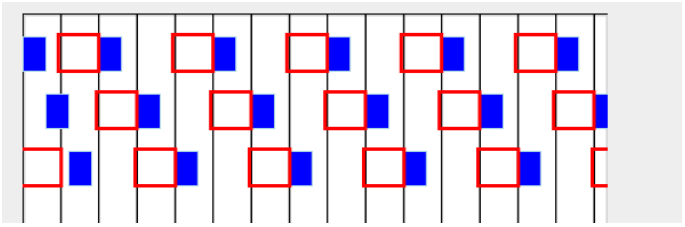
- Changing buffers is implemented using page flipping
- We assume frames are rendered in exactly 10 ms and page flipping is instant

For the subtasks, we need the length of the vsyn interval in ms, which is $\frac{1}{60}\,s = 16.\bar{6}\,ms$.

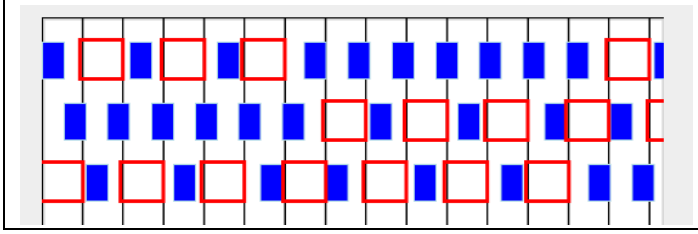
a) Whenever we finished rendering a frame, we have to wait for vsync because we may not write into the other buffer. Therefore, we always wait for $16.\bar{6} - 10 = 6.\bar{6}\,ms$.



B) In the FIFO mode, we cycle through our buffers. Even if we can initially fill up three buffers in a row without having to wait, in the long run, we will become synchronized with the vsync and again have to wait $6.\bar{6}\,ms$.



C) Since we always have one buffer that is free for writing to, we will never have to wait (= 0ms).



T2.5 Performance (1 Point)

Consider the following parts of a program.

a) Where are the NPCs allocated: On the stack or on the heap?
b) What possible performance problem can you find in the code below? Provide an alternative that is more performant.

```
// Forward declaration, defined elsewhere
class NPC;
```

```
//...
// Initialization code
const int NumNPCs = 10000;
NPC** NPCs = new NPC*[NumNPCs];
for (int i = 0; i < NumNPCs; i++)
{
      NPCs[i] = new NPC();
}


//...
// Updating all NPCs
for (int i = 0; i < NumNPCs; i++)
{
      NPCs[i]->Update();
}
```

a) When we use the operator new(), we allocate memory on the heap (unless we have overloaded it and changed this behavior).

b) The largest problem of the code stems from the fact that it calls new() each time to allocate a new NPC. This has two aspects that are sub-optimal:

- *Each time we call new(), we ask the operating system for a new memory area. This can take a long time. If we only allocated enough memory once for all NPCs in a row, we would only have to wait once for the memory allocation.*
- *Since we allocate a new memory block each time, we are not guaranteed to get the memory blocks next to each other. When we iterate over the NPCs, we will probably be jumping around in memory, leading to many cache misses and therefore more time than we would need.*

An alternative solution would use operator new[], and allocate an array of NPCs directly. This is guaranteed to be in contiguous virtual memory by the C++ standard:
*"The allocation function attempts to allocate the requested amount of storage. If it is successful, it shall return the address of the start of a block of storage whose length in bytes shall be at least as large as the requested size." (*http://eel.is/c++draft/basic.stc.dynamic.allocation*)*

For a very large array, virtual memory might still be mapped to different physical memory areas, but at least large blocks of our array will be contiguous in physical memory.

Alternative code that uses this method would be the following:

```
// Initialization code
const int NumNPCs = 10000;
NPC* NPCs = new NPC[NumNPCs];

//...
// Updating all NPCs
for (int i = 0; i < NumNPCs; i++)
{
      NPCs[i].Update();
}
```

Note: Your code doesn`t have to match exactly as long as you identified a valid performance problem and took steps to counteract it. You could also use a std::vector if you preferred it. Also, in the exam, you will not be asked to write C++ code.