Information Engineering Express
International Institute of Applied Informatics
2019, Vol. 5, No. 2, 83 - 98

High-Performance Graphics with DirectX in Racket*

Antoine Bossard †

Abstract

Parallel processing is ubiquitous with modern computer systems. It has been supported by two major hardware advances: on the one hand, computer CPUs nowadays include several processing cores – even mobile device CPUs. On the other hand, the adoption of parallel processing has been bolstered in recent years thanks to the parallel processing capacities of graphics processing units (GPUs). General-purpose computing on graphics processing units (GPGPU) is one example of the versatile capacity of GPUs. Relying on the Microsoft DirectX 12 framework, we propose in this paper a novel approach to enable parallel processing for graphical rendering on both the CPU and GPU for the functional programming language Racket (formerly PLT Scheme). Importantly, this is achieved without compromising the usability and programmer-friendliness of Racket. Significant improvements are observed from the performance evaluation experiments with respect to the execution time (×3 speed-up in some cases), the CPU utilisation time (reduced by as much as 80% in some scenarios) and the frame rate in the case of animated graphics.

Keywords: GPU, parallel processing, functional, programming, Scheme, Windows, Win32

1 Introduction

Recent advances in semiconductors have enabled the embedding of numerous processing cores into graphics processing units (GPUs), and the optimised structure of GPUs makes them suitable for parallel processing [2]. From a computer system standpoint, this leads to augmented parallel processing by concretely offloading tasks from the CPU to the graphics hardware. As of 2019, GPUs include thousands of processing cores (e.g. 2560 cores for the Nvidia GTX 1080 and 4608 cores for the superlative Nvidia TITAN RTX), compared with only a few cores for CPUs (e.g. up to 10 for the Intel Core i7 and up to 18 for the Intel Core i9, currently). So as to fully take advantage of these hardware capabilities, system vendors provide programming interfaces (APIs) to access and use the GPU, and that without having to enter the kernel mode: APIs are usable in user mode, providing indirect but secured access to the hardware. For example, Nvidia provides the CUDA parallel computing platform [3], while Microsoft has the DirectX framework.

The framework proposed in this paper relies on DirectX. It is recalled that Direct2D is built on top of Direct3D to benefit from graphics hardware acceleration, with Direct3D

* An extended abstract of this paper has been published in [1].

† Graduate School of Science, Kanagawa University. 2946 Tsuchiya, Hiratsuka, Kanagawa, Japan 259-1293.

(from Direct3D 10) standing itself on top of the DirectX Graphics Infrastructure (DXGI). DXGI is the lowest layer of the user mode and is in charge of communicating with the kernel mode (drivers) [4, 5]. Also, Direct2D provides fallback to software rendering in case hardware acceleration is unavailable. In addition, it should be noted that the software rasterizer provided by Direct2D performs significantly better than the legacy Graphics Device Interface (GDI [6], GDI+ [7]) software rendering solutions.

Our objective in this paper is to provide such a parallel processing capacity for graphics on both the CPU and GPU to the Racket (formerly PLT Scheme) functional programming language and development environment [8, 9] with DirectX. More precisely, we describe for the first time an implementation of Direct2D into Racket and quantitatively evaluate the induced performance. Also, we demonstrate the high usability of our approach compared to existing solutions. By focusing on DirectX, this research is obviously applicable to the Microsoft Windows implementation of Racket.

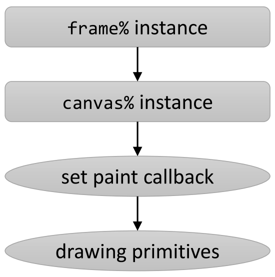
Racket provides the `<dc%>` interface so as to render 2D graphics via a device context. The merit of this approach is its simplicity. Technically, the `<dc%>` rendering approach relies on the cairo 2D graphics library (refer for instance to the Section 29 of the Racket documentation[1] and to the Racket source files `draw/unsafe/cairo.rkt`, `draw/private/bitmap.rkt` and `draw/private/dc.rkt`) which in turns relies on the legacy GDI graphics software renderer (refer for instance to the cairo source file `win32/cairo-win32-display-surface.c`). Racket also provides minimal support for 3D graphics with OpenGL. However, OpenGL's usability in Racket is low: the function bindings are merely declared, thus inducing a low usability due to complexity, especially for the users who target 2D graphics. It is definitely not a viable solution for 2D graphics such as that relying on the native `<dc%>` interface. Therefore, as of today, the Racket user is given the choice to either rely on the easy-to-use `<dc%>` interface but which performs poorly, or to switch to the overpowered OpenGL bindings which are not well suited for 2D graphics. And this is precisely to solve this dilemma for Windows native applications that Microsoft introduced Direct2D: it was too cumbersome to use Direct3D to just do 2D graphics. Thus, Direct2D, similarly to Racket's `<dc%>` interface, works in *immediate mode*, that is without handling graphics through a scene as OpenGL and Direct3D do.

The rest of this paper is organised as follows. Conventional graphics with Racket are briefly recalled in Section 2. Then, the details of our approach based on Direct2D and its access from within Racket are given in Section 3. Performance evaluation is conducted in Section 4 and the obtained results are discussed in Section 5. Finally, this paper is concluded in Section 6.

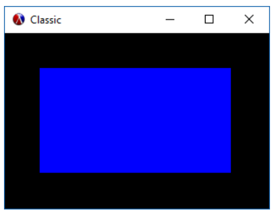
2 Preliminaries: the `<dc%>` Interface

First, we briefly recall the conventional method that is used to render graphics inside a Racket window; this is canvas painting. As explained, this method relies on the legacy GDI software renderer of the operating system. A frame (a.k.a. a window) is first created, next a canvas is added onto it. A paint callback function is attached to the canvas object, and its execution is triggered by the parent window whenever repaint is needed. Actual drawing inside the canvas is realised through the drawing context `<dc%>` interface. An instance of a device context object is passed to the paint callback function (its second parameter). Drawing commands such as `draw-rectangle` and set-brush are methods of the `<dc%>` interface. The flow of the main rendering steps is illustrated in Figure 1a.

[1] As of September 2019. See `https://docs.racket-lang.org/draw/unsafe.html`.



(a)



(b)

Figure 1: (a) The main rendering steps of the conventional method (<dc%>).
(b) Conventional window painting in Racket with the <dc%> interface.

A short code sample is given in Listing 1, with the result shown in Figure 1b.

Listing 1: Conventional window painting with the <dc%> interface.

```
(define (dc-paint canvas dc) ; paint callback
  (send dc set-pen "" 0 'transparent)
  (send dc set-brush "blue" 'solid)
  (send dc draw-rectangle 40 40 260 160))
(define classic-frame
  (new frame% [label "Classic"] [width 320] [height 240]))
(define classic-canvas
  (new canvas% [parent classic-frame] [paint-callback dc-paint]))
(send classic-canvas set-canvas-background (make-object color%))
(send classic-frame show #t) ; display window
```

3 Direct2D from within Racket

3.1 Foreign Function Interface and COM

Racket enables the programmer to call foreign functions through the Foreign Function Interface (FFI). Most notably, this library 1) enables facilitated external library loading, and 2) introduces new data types such as pointers that are originally absent from Racket. Also, C structures are available with the `define-cstruct` function. Concretely, a structure of type A is initialised with the `main-A` function which returns a pointer to the newly created structure. Importantly, memory allocation and release are automatically handled by Racket.

The DirectX subsystem is accessible through the Component Object Model (COM) [10] of the operating system. Nonetheless, DirectX enables to conveniently abbreviate COM's `CoCreateInstance` and `QueryInterface` verbose calls to instantiate Direct2D interfaces by providing the `D2D1CreateFactory` function. Effectively, directly from the interface identifier of the `ID2D1Factory` interface (i.e. IID, here of value `{06152247-6f50-465a-9245-118bfd3 b6007}`), this function returns a pointer to a `ID2D1Factory` instance, which is the gate to Direct2D interfacing. Yet, before being accessible from within Racket, the `D2D1CreateFactory` function needs to be loaded from the dynamic-linked library (DLL) `d2d1.dll`; this is realised according to the two functions of Listing 2.

Listing 2: Retrieving an instance of the ID2D1Factory COM interface.

```
(define-ffi-definer define-d2d1 (ffi-lib "d2d1.dll")) ; load library
(define-d2d1 D2D1CreateFactory ; import library function
  (_hfun _int
         (_ptr i _GUID) ; pointer to IID's value
         _D2D1_FACTORY_OPTIONS-pointer/null
         (pIFactory : (_ptr o _ID2D1Factory-pointer))
         -> D2D1CreateFactory pIFactory)) ; returns ID2D1Factory*
```

In addition, the FFI library provides helper functions for COM operations. Our implementation relies on such helper functions so as to manipulate interface instances, like declaring and calling interface methods. For example, the previously retrieved `ID2D1Factory` instance requires to first declare the corresponding interface. Since interface methods are function pointers, the method declaration order of an interface has to be strictly reproduced. To obtain such information, it is required to investigate the Windows SDK header files. The declaration of the `ID2D1Factory` interface is given as example in Listing 3.

Listing 3: Declaring the `ID2D1Factory` COM interface.

```
(define-com-interface (_ID2D1Factory _IUnknown)
  ([ReloadSystemMetrics _fpointer]
   [GetDesktopDpi _fpointer]
   ... ; declaration of other methods abbreviated
   [CreateHwndRenderTarget ; returns ID2D1HwndRenderTarget*
     (_hmfun _D2D1_RENDER_TARGET_PROPERTIES-pointer
         _D2D1_HWND_RENDER_TARGET_PROPERTIES-pointer
           (pHwndRenderTarget : (_ptr o _ID2D1HwndRenderTarget-pointer))
              -> CreateHwndRenderTarget pHwndRenderTarget)
     #:release-with-function Release]
   [CreateDxgiSurfaceRenderTarget _fpointer]
   [CreateDCRenderTarget _fpointer]))
```

The interface methods that remain unused (uncalled) in the program are simply declared as `_fpointer`. The unique method of this interface we will call is `CreateHwndRenderTarget`, whose purpose is given below. The `#:release-with-function` statement indicates that the FFI library Release method is automatically called on the resulting instance (i.e. `ID2D1HwndRenderTarget`).

Next, the factory instance is used to bind a Direct2D render target to a window. This is realised by calling the `CreateHwndRenderTarget` method of the `ID2D1Factory` interface, specifying the handle (`HWND`) of the window to be bound. Window creation in Racket has been briefly recalled in Section 2; calling the inherited `frame%` method `get-handle` returns the desired `HWND`. The other parameters are structures for settings, which include the window dimensions. In addition, the first member of the `D2D1_RENDER_TARGET_PROPERTIES` structure importantly specifies whether to use the hardware, software, or default rendering mode, the latter letting DirectX decide: hardware rendering if available, software rendering otherwise. The main result of the `CreateHwndRenderTarget` method is a pointer to a render target, precisely in this case a pointer to an instance of the `ID2D1HwndRenderTarget` interface, which inherits from `ID2D1RenderTarget`. Henceforth, all the Direct2D drawing primitives called on the render target instance will be reflected in the window; this is illustrated in the next section.

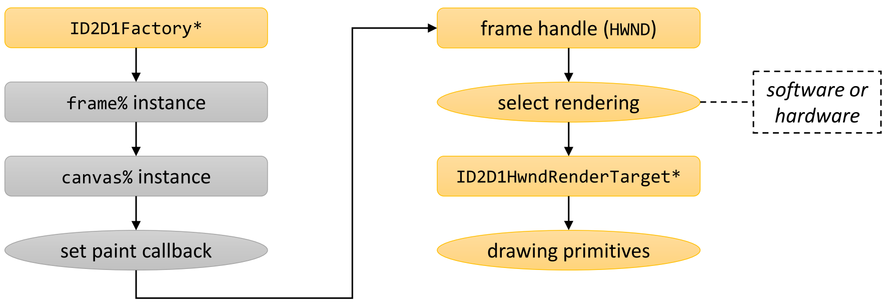


Figure 2: The main rendering steps of the proposed method (Direct2D). Racket native operations are shown in grey and DirectX ones in yellow.

The flow of the main rendering steps is illustrated in Figure 2.

3.2 Sample Application Implementation

A sample Racket application relying on Direct2D graphics is given in Listing 4; yet, COM source code as introduced earlier is abbreviated. Line 9 indicates that the first member of the `D2D1_RENDER_TARGET_PROPERTIES` structure is set to the default rendering mode, hence letting DirectX choose between hardware and software rendering. Lines 11–14 are for Direct2D resource creation, here a blue brush and an identity matrix.

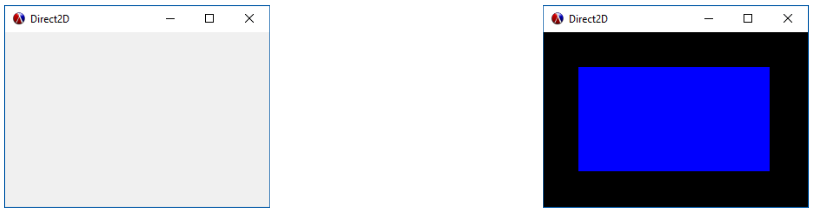
Listing 4: Sample Racket application relying on Direct2D graphics.

```
 1 (define d2d-frame
 2   (new frame% [label "Direct2D"] [width 320] [height 240]))
 3 (define d2d-hwnd (send d2d-frame get-handle))
 4 (define-values (client-w client-h) (send d2d-frame get-client-size))
 5 (define pDirect2dFactory (D2D1CreateFactory 0 IID_ID2D1Factory #f))
 6 (define pRenderTarget
 7   (CreateHwndRenderTarget
 8     pDirect2dFactory
 9     (make-D2D1_RENDER_TARGET_PROPERTIES 0 (make-D2D1_PIXEL_FORMAT 0 0)
        0.0 0.0 0 0)
10     (make-D2D1_HWND_RENDER_TARGET_PROPERTIES d2d-hwnd (make-D2D_SIZE_U
        client-w client-h) 0)))
11 (define pBlueColor (make-D3DCOLORVALUE 0.0 0.0 1.0 1.0))
12 (define pSolidColorBrush
13   (CreateSolidColorBrush pRenderTarget pBlueColor #f))
14 (define pIdentityMatrix (make-D2D_MATRIX_3X2_F 1.0 0.0 0.0 1.0 0.0 0.0))
15 (send d2d-frame show #t) ; make the window visible
```

Additionally, in Listing 5 is described the `d2d-paint` function inside which actual Direct2D operations such as drawing are conducted. As stated earlier, Direct2D works in immediate mode rendering; drawing operations are delimited by the `BeginDraw` and `EndDraw` methods of the `ID2D1RenderTarget` interface. Thus, once the window created and displayed (Figure 3a), evaluating in the Racket REPL the `d2d-paint` function updates the window as illustrated in Figure 3b.

Listing 5: Function executing Direct2D actual drawing operations.

```
(define (d2d-paint)
 (BeginDraw pRenderTarget)
 (SetTransform pRenderTarget pIdentityMatrix)
 (Clear pRenderTarget #f) ; black by default
 (FillRectangle pRenderTarget (make-D2D_RECT_F 40.0 40.0 260.0 160.0)
     pSolidColorBrush)
 (EndDraw pRenderTarget))
```



(a) (b)

Figure 3: Sample Racket application displaying an initially blank window (a), later successfully updated with Direct2D graphics (b).

3.3 Window Interaction Improvement

We have shown in the previous section with a sample application that once the window created by Racket, it is possible to emit Direct2D commands which will then update and render graphics inside the window. Yet, the best practice is to render graphics at window creation time, that is, not first displaying a blank window. More precisely, drawing commands should be executed when the application receives the operating system's `WM_PAINT` message. In Racket, as illustrated in Section 2, this is realised by adding a canvas to the window (for drawing operations), and by providing a paint callback to the canvas, the callback providing access to the device context (`<dc%>`) for drawing. This callback function would be called each time the operating system needs to repaint the window.

If we follow the same approach of adding a canvas to the window and conducting Direct2D graphical operations onto this canvas (through the canvas' `HWND` handle, a canvas being a child window), we run into the following issue. Since a canvas is double-buffered [11], any change made inside the canvas paint callback function by Direct2D, that is directly through the canvas `HWND` handle and not through its device context `<dc%>`, is ineffective as the very last operation once the paint callback function completed is the buffer swap between the back buffer and the front buffer (this buffer swap operation is an implicit finalization operation by a canvas). Hence, it is required to use the window's `HWND` handle directly as it is unfortunately not possible to rely on a canvas HWND handle as explained.

This is indeed unfortunate as, unlike for a canvas object, there is no easy way to catch the painting event for a window created in Racket. One solution to catch the `WM_PAINT` message for our Racket window is to subclass the window [12]. This can be done by using the `SetWindowSubclass` function of the `comctl32.dll` library to install a new layer of `WindowProc` callback for message catching and processing. Messages not handled by the newly installed `WindowProc` are directly forwarded to the next `WindowProc` callback by calling the `DefSubclassProc` functions. Details are given in Listing 6. Even though the Racket window was successfully subclassed to catch messages, it was at the cost of application stability and responsiveness.

Listing 6: Subclassing a Racket window to catch repaint events.

```
(define-comctl32 SetWindowSubclass ; import library function
  (_wfun _pointer
    (_wfun _pointer _uint _long _long _intptr _intptr -> _long)
    _intptr _intptr -> _int))
(define-comctl32 DefSubclassProc ; import library function
  (_wfun _pointer _uint _long _long -> _long))
(define (my-proc hWnd uMsg wParam lParam uIdSubClass dwRefData)
  (if (= uMsg 15) ; WM_PAINT message
    (begin (d2d-paint) 0) ; message processed, return 0
    (DefSubclassProc hWnd uMsg wParam lParam))) ; forward other messages
(SetWindowSubclass d2d-hwnd my-proc 0 0) ; subclass the window
```

Since the subclassing approach was not satisfactory, we instead considered to rely on a canvas for its paint callback which catches repaint messages, but forcing the canvas to stay transparent so that the double buffering issue does not erase Direct2D renderings made on the canvas' underlying window (see Figure 4). This can be achieved by specifying the `'no-autoclear` canvas style when instantiating the `canvas%` class. Yet, at window display time the canvas would still erase underlying window graphics. To address this remaining issue, one trick is to use the `'gl` canvas style in addition to `'no-autoclear`, this time completely preventing canvas underlying content erasing. Eventually, the window is created as shown in Listing 7.

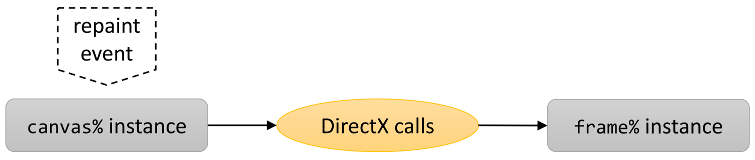


Figure 4: Catching repaint events with a canvas but directly drawing onto the window.

Listing 7: Window creation for repaint event catching combined with DirectX rendering.

```
(define d2d-frame
  (new frame% [label "Direct2D"] [width 320] [height 240]))
(new canvas% [parent d2d-frame] [style '(no-autoclear gl)]
  [paint-callback (lambda (cv dc) (d2d-paint))])
```

Another satisfactory approach is to use a canvas with the `'no-autoclear` style only and with the same paint callback, and make a trade-off to catch repaint messages with the `on-superwindow-show` message which is triggered upon window visibility changes, including the window creation event. Still, visibility change events are not identical to repaint events, hence the trade-off. The merit of this approach is its clarity: there is no `'gl` canvas style trick. This is concretely realised by deriving the `frame%` class and calling the `refresh` method to then trigger the canvas paint callback. Details are given in Listing 8.

As a result, we were able to solve all the repainting issues so that our DirectX-enabled window behaves fully as a normal Racket window, and as a normal window in general.

Listing 8: Deriving the frame% class to catch visibility change events.

```
(define d2d-frame%
  (class frame% ; base class
    (define/override (on-superwindow-show shown?) ; catch vis. changes
      (when shown? (send this refresh))) ; request a repaint
    (super-new))) ; base class constructor
```

Now, the COM interfacing, COM interface method calls and other various technical issues of the proposed Direct2D Racket system may induce anyhow computational overhead. It is thus important to evaluate the performance of the proposed system, which is the purpose of the next section.

4 Performance Evaluation

We conducted several experiments to quantitatively measure the performance of the proposed DirectX implementation and the gains made compared to the conventional `<dc%>` approach. The results were obtained with the Racket function `time` and reported as *CPU time* (i.e. actual CPU time taken to run the program), *real time* (i.e. actual execution time of the program) and *garbage collection time*. We conducted experiments with both the hardware and the software Direct2D rendering modes.

The first two experiments were conducted on a mid-range computer equipped with a 4-core (8-thread) Intel Core i7-4510U CPU (mobile processors) and its embedded GPU Intel HD 4400 (GT2) which includes 20 cores. Importantly as our implementation relies on DirectX 12, hardware acceleration is available and this GPU supports the Direct3D device driver interface (DDI) 12 and has feature levels up to `11_1`. In the third experiment, we relied in addition on a second device: a high-range computer equipped with a 4-core (8-thread) Intel Core i7-6700 CPU and an Nvidia GeForce GTX 1050 GPU which includes 640 cores. This GPU supports the Direct3D DDI 12 and has feature levels up to `12_1`.

4.1 Experiment 1 – Simple Shapes

The first performance evaluation experiment was focused on drawing simple shapes, precisely a large amount of rectangles (10,000) inside a window of size 640×480 pixels, measuring the time required for rendering. The drawing function `d2d-paint` for our Direct2D approach, and the drawing function `dc-paint` for the conventional `<dc%>` approach are given in Listing 9. Each of the two programs was run three times consecutively.

Both interpreted (i.e. run from within the Racket development environment DrRacket) and compiled (i.e. run from an executable file) versions of these experiment programs were tested. First, the results obtained with the conventional `<dc%>` approach and the Direct2D software rendering mode are given in Table 1.

Next, the results obtained with the Direct2D hardware rendering mode are given in Table 2. In this table, the results obtained with the conventional `<dc%>` approach are those from Table 1 as they are obviously not impacted by the rendering mode (software, hardware or default) of Direct2D.

Listing 9: Drawing functions for Experiment 1.

```
(define (d2d-paint)
  (BeginDraw pRenderTarget)
  (SetTransform pRenderTarget pIdentityMatrix)
  (Clear pRenderTarget #f)
  (for ([i (in-range 1 10000)]) ; 1 <= i < 10000 (integers)
    (FillRectangle pRenderTarget (make-D2D_RECT_F 0.0 0.0 (exact->inexact
        i) (exact->inexact i)) pSolidColorBrush))
  (EndDraw pRenderTarget))
(define (dc-paint canvas dc)
  (send dc set-pen "" 0 'transparent)
  (send dc set-brush "blue" 'solid)
  (for ([i (in-range 1 10000)]) (send dc draw-rectangle 0 0 i i)))
```

Table 1: Rendering time for Experiment 1 with the conventional `<dc%>` approach and with the proposed approach (Direct2D software rendering mode). Units: milliseconds (the lower the better).

	Conventional (`<dc%>`)			Direct2D (software)		
	CPU	real	gc	CPU	real	gc
interpreted	766	770	31	735	216	0
	750	748	16	782	201	0
	718	717	0	766	201	0
compiled	688	718	0	750	216	0
	718	720	0	750	200	0
	703	702	0	812	201	0

4.2 Experiment 2 – Animation

While Experiment 1 focused on the rendering of still simple shapes, in this second experiment, we measure the performance of our approach in the case of animations. Concretely, using the same machine as in Experiment 1, we render numerous small shapes in a grid fashion, and the animation consists in rotating the whole by 0.1 radian at each frame. The performance is quantitatively measured with the frame per second (FPS, a.k.a. frame rate) metric which is commonly used by various graphics benchmarks.

More precisely, the animation took place inside a window of size 640×480 pixels and involved 8×8 pixels circles, filled according to a linear gradient brush horizontal and 100 pixels wide, with three stops: red at 0.0, green at 0.5 and blue at 1.0. Anti-aliasing was used (default anti-aliasing for Direct2D, and `set-smoothing` set to `'smoothed` for the device context of the conventional `<dc%>` approach). The drawing functions used in the `<dc%>` and Direct2D cases are shown in Listing 10. It can be observed that the two functions are as similar as possible for fair evaluation. In the `<dc%>` case, the translation transformation is already applied as preprocessing, hence not appearing in the drawing function (this can thus be seen as a slight performance disadvantage for our approach; we will see that our approach nonetheless beats clearly the conventional one).

The animation loop in the `<dc%>` case is given in Listing 11. It is exactly the same for the

Table 2: Rendering time for Experiment 1 with the conventional `<dc%>` approach and with the proposed approach (Direct2D hardware rendering mode). Units: milliseconds (the lower the better).

	Conventional (`<dc%>`)			Direct2D (hardware)		
	CPU	real	gc	CPU	real	gc
interpreted	766	770	31	125	804	0
	750	748	16	172	782	0
	718	717	0	172	788	0
compiled	688	718	0	172	760	0
	718	720	0	110	776	0
	703	702	0	94	808	0

Listing 10: Drawing functions for Experiment 2.

```
(define (dc-paint canvas dc)
  (send dc clear)
  (send dc rotate 0.1)
  (for* ([x (in-range -320 320 10)] [y (in-range -240 240 10)])
    (send dc draw-ellipse x y 8 8)))

(define (d2d-paint)
  (BeginDraw pRenderTarget)
  (Clear pRenderTarget #f)
  (mat32-multiply! pMatrix pRotMatrix) ; rotation
  (SetTransform pRenderTarget
    (mat32-multiply pMatrix pTranslateMatrix)) ; translation
  (for* ([x (in-range -320 320 10)] [y (in-range -240 240 10)])
    (FillEllipse pRenderTarget (make-D2D1_ELLIPSE (make-D2D_POINT_2F
        (exact->inexact x) (exact->inexact y)) 4.0 4.0) pGradientBrush))
  (EndDraw pRenderTarget))
```

Direct2D case, except that `d2d-paint` is called instead of `dc-paint`, and `d2d-frame` instead of `classic-frame`. A screenshot of the animation is given in Figure 5. The average frame rates for the first 50 frames of the animation are given in Table 3.

Listing 11: Animation loop for Experiment 2.

```
(define (run-classic)
  (send classic-frame show #t)
  (let run ([t (current-milliseconds)])
    (yield)
    (dc-paint classic-canvas dc)
    (let ([t2 (current-milliseconds)])
      (displayln (/ 1 (* 0.001 (- t2 t)))) ; FPS value
      (run t2))))
```

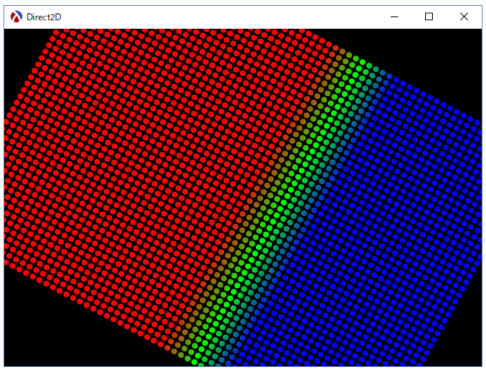


Figure 5: Screenshot of the rotating animation in Experiment 2.

Table 3: Measurement of the average frame rate and flickering for Experiment 2 with the conventional `<dc%>` approach and with the proposed approach (Direct2D). Units: frame per second (the higher the better). The standard deviation is given in parentheses.

	Conventional (`<dc%>`)	Direct2D	
		software	hardware
Average FPS	6 (1)	38 (73)	61 (17)
Flickering	severe	none	none

4.3 Experiment 3 – The Koch Snowflake

A third evaluation experiment was conducted so as to confirm that the obtained results were not depending on the experimental conditions, precisely the computer used for Experiments 1 and 2. The technical specifications of the two computers used in this experiment were detailed at the beginning of Section 4. In addition, this third experiment tested another usage scenario: procedural graphics, with animation. Concretely, we assembled a program generating and rotating a Koch snowflake – at any generation step – with our Racket implementation of DirectX.

In practice, after specifying the number of steps for the Koch snowflake, the scene was rotated by 0.1 radian at each frame. As in Experiment 2, the performance was measured through the smoothness of the resulting animation, relying on the frame per second metric. The Koch snowflake was initialised with a side of 300 pixels (i.e. the first generation step of the Koch snowflake induces an equilateral triangle of side 300 pixels) inside a 640×480 pixels window. Regarding the functions defined for generating the fractal, they are rigorously identical, with in the conventional `<dc%>` approach calling the `draw-line` method of the `<dc%>` class for drawing a straight line, while calling the `DrawLine` method of the `ID2D1RenderTarget` interface in the DirectX approach. The source code used for this experiment is given in Appendix A. The obtained Koch snowflake at generation step 4 when rendering with DirectX is illustrated in Figure 6 (experiments were conducted with the default background colour, black, greyed in this figure only for printing matters).

The average frame rates obtained from the first 50 frames under the various experimental conditions are given in Table 4 for the mid-range computer, and in Table 5 for the high-range computer. Both results for the DirectX software and hardware rendering modes are shown.

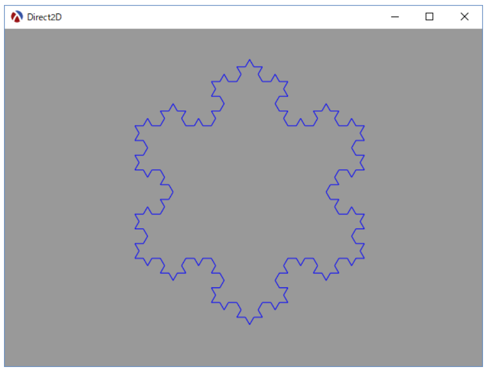


Figure 6: DirectX-rendered Koch snowflake at generation step 4.

Table 4: Average frame rate for Experiment 3 with the conventional `<dc%>` approach and with the proposed approach (Direct2D) on the mid-range computer. Units: frame per second (the higher the better). The standard deviation is given in parentheses.

k	Conventional (`<dc%>`)	Direct2D (software)	Direct2D (hardware)
4	637 (450)	804 (373)	125 (226)
5	119 (227)	606 (453)	99 (185)
6	16 (3)	74 (139)	67 (49)
7	4 (0)	16 (5)	15 (4)
8	1 (0)	4 (0)	3 (0)
9	0 (0)	1 (0)	1 (0)

When less than 1 ms was taken between two consecutive rendered frames, the FPS was set to 1000 FPS, which is a lower bound on the frame rate and corresponds to exactly a 1 ms delay between two frame renderings. The results are shown function of k, the step at which the Koch snowflake was generated and rendered (as mentioned earlier, the first step $k = 1$ induces the generation of an equilateral triangle). Hence, the decreasing frame rate as the value of k increases. At $k = 9$, the animation is almost stopped (0 FPS for the conventional approach, 1 FPS for the proposed approach).

Table 5: Average frame rate for Experiment 3 with the conventional `<dc%>` approach and with the proposed approach (Direct2D) on the high-range computer. Units: frame per second (the higher the better). The standard deviation is given in parentheses.

k	Conventional (`<dc%>`)	Direct2D (software)	Direct2D (hardware)
4	798 (369)	848 (300)	144 (261)
5	360 (439)	521 (373)	182 (280)
6	25 (6)	321 (424)	136 (192)
7	7 (1)	22 (7)	19 (4)
8	2 (1)	5 (1)	5 (1)
9	0 (0)	1 (0)	1 (0)

5 Results Discussion

In this section, we discuss the experimental results obtained in Section 4, investigating about the gains induced by the proposed approach.

5.1 Experiment 1

First, we notice from the results of Experiment 1 that there is no significant performance difference between the interpreted and compiled versions of the programs. Then, it is important to observe that the conventional approach induces the longest (or near-longest) times in both CPU time and real time. Now, when using Direct2D in software mode, the CPU time stays similar to that of the conventional approach (an average time increase of+6% for the Direct2D approach with respect to the `<dc%>` approach) while the real time is reduced by about 70% (an average time decrease of 72% for the Direct2D approach with respect to the `<dc%>` approach), which is a first significant positive result. That the CPU time is not improved is no surprise since the rendering, even though using Direct2D, is done in software mode, that is without any support from the GPU hardware. Hence, the time taken by the CPU in total is similar to the `<dc%>` approach. The execution time (real time) significant improvement can be explained by the multicore architecture of the CPU used for the experiment.

Regarding the hardware rendering experiment, this is the reversed situation. Effectively, while the execution time (real time) remains similar to that of the conventional approach (an average time increase of +8% for the Direct2D approach with respect to the `<dc%>` approach), the CPU time taken in total by the hardware-accelerated Direct2D program is reduced by about 80% (an average time decrease of 81% for the Direct2D approach with respect to the `<dc%>` approach), which is a second significant positive result. In Direct2D hardware rendering mode, graphics-related processing is mainly conducted on the GPU, so the total CPU time taken stays very low. Yet, the execution time remains high since the GPU used in this experiment is low-end (CPU embedded), and the graphics involved are simple. Still, being able to achieve such a CPU usage reduction (balanced with GPU usage obviously) is an important achievement for at least two reasons:

- CPU time is made available for other system applications, especially those that do not rely on the GPU (they are a majority).
- CPUs have a much lower GFLOP/Watt ratio than GPUs [14], especially when consid-ering the GFLOP per second per processing element (core) ratio [13], and are thus from this point of view much less environment-friendly (see Green 500 [15]) than GPUs.

5.2 Experiment 2

The significantly positive results of Experiment 1 were confirmed by Experiment 2. Ef-fectively, from the frame rate measurements obtained in the second experiment, it is easy to assess the large performance increase when using our Direct2D approach compared to the conventional `<dc%>` approach. We observe a jump from an average 6 FPS for the `<dc%>` approach to full animation fluidity with the Direct2D approach. Also, when measuring the average frame rate of the first 50 frames, we noticed a 40% FPS increase when using the hardware mode of Direct2D compared to its software mode.

Qualitatively, it is also important to mention that the conventional `<dc%>` approach produced severe flickering during the animation.

5.3 Experiment 3

The results of Experiment 3 first confirmed that the results obtained in the previous experiments were not depending on the experimental setup: we were able to reproduce a similar speed-up between the conventional `<dc%>` approach and the proposed DirectX approach, commonly converging towards an about ×3 frame rate increase between the two approaches – in favour of the proposed approach – as the generation step of the Koch snowflake increases (especially starting with generation step $k = 6$).

In addition, although the frame rates induced by the DirectX software and hardware modes converge as the complexity of the rendered graphics increases, it can be observed that when dealing with early generation step Koch snowflakes, the DirectX hardware mode trails the DirectX software mode in terms of frame rate. This might be explained by the fact that, unlike in Experiment 2, the rendered graphics are produced by a recursive process and it is well-known that recursive algorithms are difficult to optimise with parallel processing such as that provided by GPUs. And, it is likely that as the fractal generation step gets larger, the computational complexity, rather than the graphics workload, becomes dominant, thus masking the DirectX software and hardware mode performance difference.

Finally, the standard deviation is rather large for the early generation steps of the fractal. This can be explained by the fact that side processes, especially garbage collection which occurs at regular time intervals, are very likely to impact the frame rate: since k is low, each frame is drawn almost instantly but when garbage collection occurs, the duration between the frame before and the frame after the garbage collection will be significantly larger than between other frames (i.e. when no garbage collection occurs). This side effect might be reduced in hardware mode as a larger part of the calculations would be done in parallel by the GPU, thus possibly less impacted by garbage collection since, even partially, running in a different process. This would explain the lower deviation of the average frame rate when running in Direct2D hardware mode.

6 Conclusions

Nowadays, modern computer systems rely heavily on parallel processing. Furthermore, with advances in GPU multicore technologies, parallel processing has become essential for graphical rendering. We have proposed in this paper a novel approach to enable parallel processing for graphical rendering with the Racket functional programming language and development environment. We showed 1) that it is possible to use DirectX within a Racket program, even interpreted; 2) that it is easy: we proposed an arguably elegant implementation; and 3) that it gives excellent results thanks to the parallel processing capacities of the CPU and GPU chips. Precisely, we managed to reduce the CPU time for graphics by more than 80% in average with the Direct2D hardware rendering mode compared to the legacy approach. Also, we obtained another significant result: by using the Direct2D software rendering mode instead of the legacy `<dc%>` approach of Racket, we were able to achieve in average a ×3 execution speed-up. And, we confirmed that the animation frame rate was significantly higher when using Direct2D instead of `<dc%>`, completely eliminating flickering, and that the obtained results did not depend on the experimental conditions such as the computer hardware.

Regarding future works, now that we have shown that DirectX is accessible in Racket and that Direct2D produces excellent results, it would be interesting to look into Direct3D in Racket, comparing the obtained performance with that of the Racket native OpenGL bindings. Also, it would be insightful to experiment a similar approach with other software development solutions not natively including DirectX support. Lastly, accessing Nvidia's CUDA from within Racket is another meaningful future work.

Acknowledgements

The author is deeply appreciative of the insightful review comments and suggestions. This research was partly supported by a Grant-in-Aid for Scientific Research (C) of the Japan Society for the Promotion of Science under grant no. 19K11887.

References

[1] Antoine Bossard, "High-performance graphics in Racket with DirectX", Proceedings of the 17th International Conference on Algorithms and Architectures for Parallel Processing, pp. 814–825, Helsinki, Finland, 2017.

[2] Nvidia, GeForce GTX 1080 user guide, 2016.

[3] Nathan Whitehead and Alex Fit-Florea, "Precision & performance: floating point and IEEE 754 compliance for Nvidia GPUs", Nvidia, 2016.

[4] Frank D. Luna, Introduction to 3D game programming with DirectX 12 (Chapter 4), Mercury Learning and Information, Dulles, VA, USA, 2016.

[5] Microsoft, "DXGI", MSDN. `https://docs.microsoft.com/en-us/windows/win32/direct3ddxgi/dx-graphics-dxgi`. Last accessed September 2019.

[6] David Iseminger, Microsoft Win32 developer's reference library – Microsoft Windows GDI, Microsoft, WA, USA, 2000.

[7] Mahesh Chand, Graphics programming with GDI+, Addison-Wesley Professional, 2003.

[8] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler and Matthias Felleisen, "DrScheme: a programming environment for Scheme", Journal of Functional Programming, Vol. 12, No. 2, pp. 159–182, 2002.

[9] Matthew Flatt, "Creating languages in Racket", Communications of the ACM, Vol. 55, No. 1, pp. 48–56, 2012.

[10] Don Box, Essential COM, Addison-Wesley Professional, Boston, MA, USA, 1998.

[11] R. B. Sheeparamatti, B. G. Sheeparamatti, Manjula Bharamagoudar, Nayan Ambali, "Simulink model for double buffering", Proceedings of the 32nd Annual Conference on IEEE Industrial Electronics, pp. 4593–4597, Paris, 2006.

[12] Microsoft, "Subclassing controls", MSDN. `https://docs.microsoft.com/en-us/windows/win32/controls/subclassing-overview`. Last accessed September 2019.

[13] Karl Rupp, “CPU, GPU and MIC hardware characteristics over time”, 2013–2016, `https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time`. Last accessed September 2019.

[14] Vincent Hindriksen, “Processors that can do 20+ GFLOPS per Watt”, 2012, `https://streamhpc.com/blog/2012-08-27/processors-that-can-do-20-gflops-watt/`. Last accessed September 2019.

[15] Wu-chun Feng and Kirk Cameron, “The green500 list: encouraging sustainable supercomputing”, Computer, Vol. 40, No. 12, pp. 50–55, 2007.

A Experiment 3 Source Code

We give in Listing 12 the source code used in Experiment 3 to generate a Koch snowflake at step *k*. The fractal is generated by calling the koch function with the parameters being the generation step, the rendering target, coordinates and the initial side width.

Listing 12: The functions used to generate the Koch snowflake in Experiment 3.

```
(define (koch-iterate k dc x1 y1 x2 y2)
  (if (= k 1)
    (send dc draw-line x1 y1 x2 y2) ; 'DrawLine' method if DirectX.
  ; else create a thorn, and recursion
  (let* ([k (sub1 k)]
         [cos60 (cos (/ pi 3))]
         [sin60 (sin (/ pi 3))]
         [x3 (+ x1 (* 1/3 (- x2 x1)))]
         [y3 (+ y1 (* 1/3 (- y2 y1)))]
         [x5 (+ x1 (* 2/3 (- x2 x1)))]
         [y5 (+ y1 (* 2/3 (- y2 y1)))]
         [x4 (+ x3 (- (* (- x5 x3) cos60)
                      (* (- y5 y3) sin60)))]
         [y4 (+ y3 (+ (* (- x5 x3) sin60)
                      (* (- y5 y3) cos60)))])
    (koch-iterate k dc x1 y1 x3 y3)
    (koch-iterate k dc x3 y3 x4 y4)
    (koch-iterate k dc x4 y4 x5 y5)
    (koch-iterate k dc x5 y5 x2 y2))))

(define (koch k dc x y w)
  (let* ([cos60 (cos (/ pi -3))]
         [sin60 (sin (/ pi -3))]
         [x2 (+ x w)]
         [y2 y]
         [x3 (+ x (- (* (- x2 x) cos60)
                     (* (- y2 y) sin60)))]
         [y3 (+ y (+ (* (- x2 x) sin60)
                     (* (- y2 y) cos60)))])
    (koch-iterate k dc x y x2 y2)
    (koch-iterate k dc x2 y2 x3 y3)
    (koch-iterate k dc x3 y3 x y)))
```