Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Videó átméretező megvalósítása FPGA-val

Önálló laboratórium 1

Készítette
Gilicze Márton

Konzulens
Szántó Péter

2024. június 17.

Feladatkiírás

Az átméretezés igénye számos videó feldolgozó folyamatban merül fel igényként, s mivel a videó feldolgozás FPGA-ban tipikusan jól megvalósítható feladat, így ebben a környezetben is szükség van az ezt a feladatot ellátó egységre.

Ugyan mindkét nagy FPGA gyártó (Intel, Xilinx) rendelkezik ehhez gyári megoldással, azonban ezek az IP magok meglehetősen drágák, és utóbbi gyártó esetén ráadásul meglehetősen rosszul működik. A feladat célja egy saját, általánosan használható átméretező IP kifejlesztése.

A cél egy olyan blokk kifejlesztése, amely testzőleges felbontásról képes bármilyen másik tetszőleges felbontásra skálázni magasabb fokszámű szűrők használatával, azaz jó minőségben.

Tartalomjegyzék

1. fejezet

Bevezetés

A feladat során egy olyan videó átméretező IP blokk fejlesztését kezdtem meg, amely tetszőleges méretről tetszőleges méretre tud átméretezni videófolyamot. A videó átméretezésnek több alkalmazása is lehet, pl retro konzolok képeinek upscalelése nagyobb felbontásra, de egy érdekesebb felhasználása az a mesterséges intelligenciára épülő képfeldolgozásban van, ugyanis sok ilyen modell csak bizonyos méretű képeket/videókat képes bemenetként elfogadni, és ha pl a kameránk nem ilyen videófolyamot készít, akkor az mindenképpen preprocessingre szorul.

Azért választottam ezt a feladatot, hogy jobban megismerkedjek az FPGA-kal, és a digitális tervezéssel, hiszen eddigi tanulmányaim során nem találkoztam még velük. Éppen ezért a félév első felében főleg ismerkedtem a verilog nyelvvel, és a digitális tervezéssel, közben irodalomkutatást végeztem. A félév második felében pedig elkezdtem az implementációt.

A félév során azt a célt tűztem ki, hogy képeket legyen képes a modul tetszőleges méretűre átméretezni. Azzal, hogy a bemenő képfolyam milyen módon érkezik meg az FPGA-ba, jelenleg nem foglalkoztam.

Első lépének irodalomkutatást végeztem képek átméretezéséről. Itt különböző átméretezési algoritmusokat ismertem meg, melyek közül a bilineáris transzformációval, és a polyphase transzformációval foglalkoztam részletesebben.

Második lépésnek megpróbáltam egy egyszerűbb bilineáris algoritmust megvalósítani, amely tetszőleges méretről tetszőleges méretűre tud képeket átméretezni.

Harmadik lépésnek létrehoztam egy sorbuffer modult testbenchel, amely a megfelelő sorokat képes eltárolni egy bejövő folyamból, és a megfelelő pixeleket tudja adagolni az átméretező algoritmusnak.

Végezetül ezeket integráltam és test benchet hoztam létre nekik.

2. fejezet

Irodalomkutatás

2.1. Létező megoldások keresése

Első lépésnek megpróbáltam már létező videó átméretező IP-ket keresni az interneten. Az Intel[1]-nek és Xilinx[2]-nek vannak már meglévő megoldásai, azonban ezek nem működnek mindig, sokszor hibásak, és van amit már nem is támogatnak. A két nagy gyártó saját scaler-jén kívül találtam harmadik féltől való megoldást is[3], de ez is meglehetősen drága, úgyhogy egy saját IP kifejlesztésének igenis létjogosultsága van.

2.2. Kép átméretezés

Első lépésnek irodalomkutatást végeztem a képátméretezési technikákról. A képátméretezési technikák 2 fő csoportba oszthatóak: forward mapping és backward mapping.

2.2.1. Forward mapping

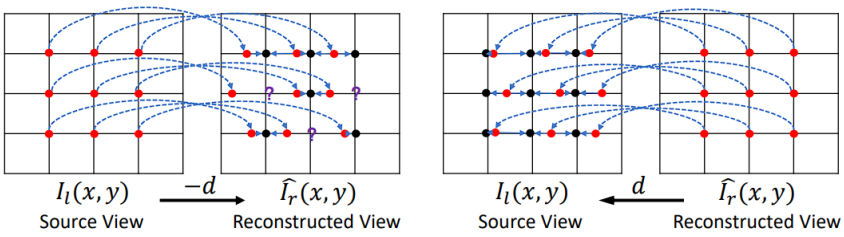
A forward mapping során egy bemeneti képnek a pixeleit próbáljuk közvetlenül átmappolni a kimeneti képnek a pixeleire. Ezt a 2.1 ábrán láthatjuk. A kimeneti képen a képpontok ilyenkor nem fognak minden esetben pontosan ráesni egy-egy pixelre, ami miatt lyukak keletkezhetnek a kimeneti kép pixelei között. Erre egy lehetséges megoldás az, hogy az eredeti lép pixeleinek éleit vesszük, és ezt a formát átalakítjuk a kimeneti képre, majd ezután egy pixel értékét annak alapján állítjuk be, hogy az eredeti pixel értéke milyen mértékben van jelen ott. [7]

Ezek miatt belátható, hogy nem olyan szerencsés ezt a stratégiát használni. Sokkal jobb lenne, ha a kimeneti kép pixeleit mappolnánk át a bemeneti kép pixeleire, hiszen így biztosan minden kimeneti pixelre esne valamilyen érték. Ezt hívjuk backwards mappingnak.

2.2.2. Backward mapping

A backward mapping során fordítva, a kimeneti kép pixeleit feleltetjük meg a bemeneti kép pixeleihez. Ekkor ugyan szintén fenn áll a forward mapping során levő probléma, miszerint a pixelek nem fognak egy-egy pixelre ráesni a bemeneti képen, de ez itt nem probléma, hiszen itt minden pixel értékét ismerjük, tehát approximálhatjuk a nem egész koordinátán levő pixel értékét valamilyen algoritmus szerint. A 2.1 ábrán egy összehasonlítás látható a backward és forward mapping működése között.

A projekt során a backward mapping megoldást használtam, hiszen ennek felhasználásával, könnyen lehet a kimeneti kép pixeleit a bemeneti képen való elhelyezkedésük



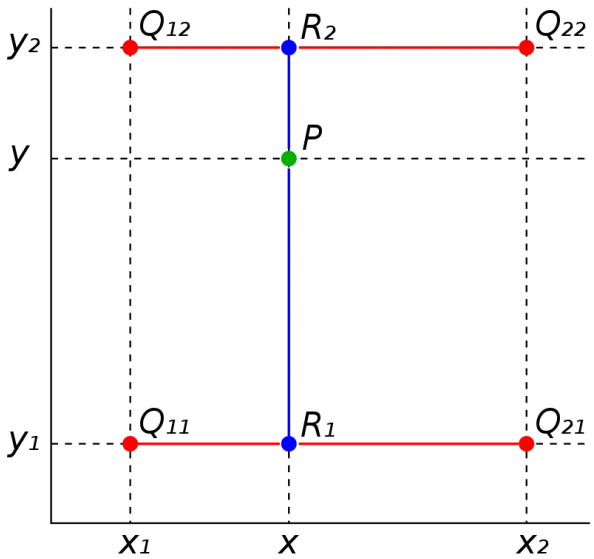
(a) Foward mapping (b) Backward mapping

2.1. ábra. A backward és forward mapping összehasonlítása [6]

alapján valamilyen algoritmus segítségével meghatározni. Most tekintsünk meg néhány ilyen átméretező algoritmust.

2.3. Bilinear filter

A bilineáris szűrő az egyik egyszerűbb átméretezési technika. A lényege az, hogy a backward mapping során egy kimeneti pixelnek meghatározzuk a koordinátáját a bemeneti képen. Ez vagy ráesik egy pixelre, vagy nem. Ha nem esik rá egy pixelre teljesen, akkor viszont approximálni lehet az értékét a környező 4 pixel értékéből. A bilineáris interpoláció, mint a neve is mondja, 2 lineáris interpolációt végez el, vízszintesen, meg függőlegesen. A súlyok ilyenkor a környező pixelektől levő távolságok lesznek. Tehát egy 1D-s esetet nézve, ha a bal oldali pixelhez 2-szer olyan közel van a pixel mint a jobb oldalihoz, akkor a súlyok 0.66 és 0.33 lesznek. [5] A 2.2 ábrán látható a vizualizációja az interpolációknak.



2.2. ábra. A bilineáris filter működése [5]

A gyakorlatban 3 interpolációt kell megvalósítani, amiből a vízszintes irányban levő 2 interpolációt egyszerre, és utána pedig a függőleges interpolációt lehet elvégezni, ami már a kimenete lesz a filternek. A legegyszerűbb módon az alábbi képlettel lehet egy interpolációt elvégezni, mondjuk az alsó 2 pixelen:

$$f(x, y_1) = \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \tag{2.1}$$

Azonban azt látjuk, hogy ez a művelet osztást igényel, amit érdemes lenne elkerülni. Mivel biztosan tudjuk, hogy ez a 4 pixel szomszédos, emiatt a nevezőkben levő különbségek biztosan 1 lesznek. Az osztás tehát eltűnt. További egyszerűsítés, hogy az $x - x_1$ és a $x_2 - x$ különbségeket könnyen tudjuk ábrázolni, hiszen a az $x - x_1$ az csak az x-nek az egész része, tehát ha fix pontosan ábrázoljuk a pixel koordinátáit, akkor csak a kettedes pont utáni biteket kell venni. Az $x_2 - x$ pedig az előző számnak a komplementere, hiszen a kettő összege garantáltan 1. Így tehát ha meghatároztuk a kimeneti pixel koordinátáját a bemeneti képen, a bilineáris filterhez szükséges együtthatókat is könnyen meg tudjuk határozni.

Innen már csak a szorzások vannak hátra, ami az előbb taglaltak miatt összesen 6 szorzás lesz, hiszen egy lineáris approximációhoz 2 szorzás kell. Ezek alapján az egyenletek a következőek lesznek:

$$R_2 = Q_{12}\,(X_2 - X) + Q_{22}\,(X - X_1) \tag{2.2}$$
$$R_1 = Q_{11}\,(X - X_2) + Q_{21}\,(X - X_1) \tag{2.3}$$
$$P = R_2\,(Y_1 - Y) + R_1\,(Y - Y_2) \tag{2.4}$$

Ha ezt az algoritmust alkalmazzuk a kimeneti kép összes pixelére, akkor egy egész jó minőségű képet kaphatunk. Az algoritmus további előnye, hogy tetszőleges átméretezést tud megvalósítani, hiszen a kimeneti kép pixelei, csak a környező 4 pixeltől függ a bemeneti képen.

2.3.1. Polyphase filter

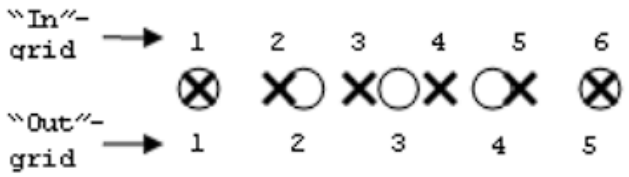
A polyphase filter, egy bonyolultabb szűrő architektúra, aminek a segítségével jobb minőségű képet tudunk generálni a bilineáris filterhez képest. A filter működése több dologban különbözik a bilineáris filtertől.

Az első ilyen különbség, hogy a polyphase szűrő az nem a környező 4 pixel értékéből dolgozik, hanem tetszőleges mennyiségű körrnyező sort/oszlopot tud felhasználni a kimeneti pixel értékének meghatározásához. A bilineáris filternél 2 sort és 2 oszlopot használtunk, tehát mint vertikális, mind horizontális irányban 2-2 "tap" je van a bilineáris szűrőnek. A polyphase filter-t sok esetben 4 vertikális és 4 horizontális tap-el alkalmazzák, tehát horizontális és vertikális irányban is 4-4 pixelt, azaz a környező 16 pixelt veszik figyelembe a kimenet meghatározásához.

A második fontos különbség az az együtthatók előállítása. A bilineáris filternél a kimeneti pixel közvetlen helyzetéből van meghatározva a környező 4 pixel súlya az alapján, hogy milyen távol van a pixel ezektől. A polyphase szűrőnél ezzel ellentétben előre le vannak generálva az együtthatók, valamilyen algoritmus alapján. Ez az algoritmus általában a bemeneti és kimeneti kép arányszámát használja fel együttható generálásra. A Xilinx ajánlása alapján [4] (48 oldal) az úgynevezett Lánczos algoritmus segítségével lehet előállítani ilyen együtthatókat.

Ezen kívül fontos megjegyezni, hogy az algoritmus nem csak (4x4 tap esetén) említett 16 algoritmust használja. Ugyanis ahogy a szűrő neve mondja, a kimeneti pixel bemeneti

képen elhelyezkedő képpont "fázisa" alapján választja ki a megfelelő együtthatókat. Nézzük meg egy egyszerű példán a működést, 1 dimenzióban az egyszerűség kedvéért. Tegyük fel, hogy a bemeneti sorban 6 pixel van, és a kimeneti sorban pedig 5. Ekkor ha megfeleltetjük a kimeneti pixeleket a bemeneti képen, akkor az 2.3 ábrát fogjuk kapni ("O"-kimeneti pixel pixel, "X" bemeneti pixel)

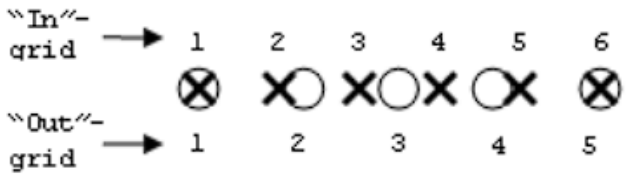


2.3. ábra. A polyphase filter működése, 1 dimenzióban [4]

Mint látjuk, a kimeneti pixel 4 féle fázisban lehet. Ezek a fázisok 0, 1/4, 2/4, 3/4. Ezeket úgy kaptuk meg hogy elosztottuk a kimeneti és bemeneti kép méretét (technikailag a kimenet-1 és a bemenet-1 méreteket), mellyel 5/4 -et kaptunk, majd ezt mindig hozzáadtuk az előző pixel értékéhez, és így megkaptuk a kimeneti pixelek koordinátáit, a bemeneti képen: $0, 5/4, 10/4, 15/4, 20/4$, melyekből már látszanak az előbb megállapított fázisok. Ekkor tehát szűrő az alapján, hogy milyen fázisban van, más együtthatókat használ a környező pixelek súlyozására. Éppen ezért a polyphase szűrő nem együtthatókkal, hanem úgynevezett coefficient bank-al rendelkezik.

2.3.2. Separable filter

A polyphase szűrő egyszerűbb, és erőforrás takarékosabb megvalósításához érdemes a separable, vagyis úgynevezett szétválasztható szűrők fogalmának ismertetése. Ez csupán annyit takar, hogy pl a bilineáris szűrő megvalósításához hasonlóan, a szűrőnket szétdaraboljuk több, 1 dimenziós interpolációra. Ez a polyphase szűrőnél azt fogja jelenteni, hogyha pl 4-4 tap-es szűrőről van szó, akkor először a 4 oszlopon hajtunk végre egy interpolálást, amiből kapunk 4 értéket, majd végül ezen 4 értéken vízszintes irányban hajtunk végre egy interpolálást, ezzel egyszerűsítve a filter architektúráját.[**?**] Ezt a mőködést a **??** ábra szemlélteti.



2.4. ábra. Egy separable polyphase filter működése

Fontos megjegyezni, hogy nem minden polyphase filter szétválasztható, ez csak akkor teljesül, ha olyan együtthatókat generálunk, amelyek erre képesek.

3. fejezet

Bilineáris filter megvalósítása

Mivel ez volt az első félév ahol FPGA-val, és a verilog nyelvvel találkoztam, ezért első lépésnek azt tűztem ki célul, hogy megalkossam a bilineáris szűrőt. Ez elég sok trial and error-t tartalmazott, mert nem csak a digitális design, és a Vivado fejlesztői környezet, hanem a verilog nyelv is új volt számomra.

Elsőnek egy olyan egyszerű modult alkottam meg, amelynek ha megadjuk a kimeneti és bemeneti kép méreteit, akkor képes meghatározni mindegyik kimeneti pixelnek a bemeneti képen a helyzetét. Ezután ezt a helyzetet outputként kiadja a modul, és a testbench visszaadja a 4 környező pixelt. Ezek után a 4 egymás melletti pixelből a modul előállítja a bilineáris filterhez szükséges súlyokat, majd végül elvégzi a megfelelő szorzásokat, és kiadja outputnak a kimeneti pixel értékét.

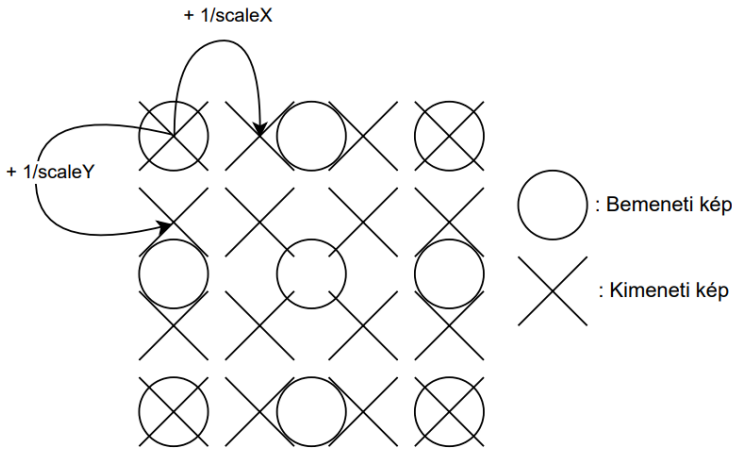
Később ez a modul bővítve lett, hogy a bemenetre azt is meg lehessen adni, hogy hány csatornán érkeznek be a pixelek. Ha ez az érték 1, akkor szürkeárnyalatos képről beszélünk, és ha ez az érték 3, akkor színes, RGB képről beszélünk. Ezen kívül egy állapotgépbe rendeztem a modult, hogy a későbbi fejezetben bemutatott sorbufferrel szinkronban tudjon működni a bilineáris filter.

3.0.1. Számláló modul

A Eddig sokat beszéltünk a backwards mappingról és különböző filter algoritmusokról, de arról nem, hogy hogyan is határozzuk meg a kimeneti kép pixeleit a bemeneti képen. Ezt a következő módon valósítottam meg. Raszter iránynak megfelelően, balról jobbra és fentről lefele, egyesével végigmentem a kimeneti kép pixelein. Ez egyszerűen megvalósítható, csak 2 számlálót kell inkrementálni, hogyha az x irányú számláló eléri a sornak a végét, akkor azt reseteljük, és inkrementáljuk az y irányú pixel számlálót. Ha az is eléri az kimeneti kép Y nagyságát, akkor készen vagyunk a képpel, ezeket resetelhetjük. Ezzel szinkronban, 2 másik számlálót is működtetünk, amely a mebemeti képen levő helyzetét számolja a kimeneti pixeleknek. Mindig amikor x irányban inkrementáljuk a kimeneti pixel számlálót, akkor az x irányú bemeneti pixel számlálót is inkrementáljuk a kimeneti és bemeneti képek arányaival. Mivel a pixelek számlálását 0 tól kezdjük, ezért ez az arányszám a következő lesz:

$$scaleX = \frac{outputX - 1}{inputX - 1}, \quad és \quad scaleY = \frac{outputY - 1}{inputY - 1} \tag{3.1}$$

Tehát a bemeneti és kimeneti képek méretének arányszáma az az érték, amivel inkrementálni kell a bemeneti kép pixel számlálóját. Ezt a működést a 3.1 ábra szemlélteti.

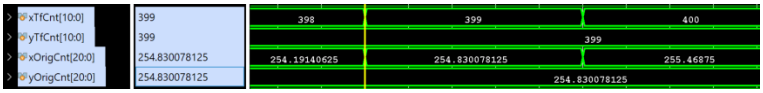


3.1. ábra. A bilineáris filter számlálóinak működése

Itt viszont észrevesszük, hogy ez az arány szám nem mindig egész szám. Éppen ezért ezt az arányszámot, és a bemeneti pixel számlálót is fix-pontos számként kell ábrázolnunk. Itt meg kellett ismerkednem a fix pontos számábrázolással, ami alapján sikerült is ezt a számláló egységet megvalósítanom. A számábrázolás során fontos volt, hogy a fix pontos számok maximum 18 biten legyenek ábrázolva, hiszen általában egy FPGA-ban ekkora bites számok szorzására képesek a LUT szorzók.

Fontos megjegyezni, hogy ezeket az arányszámokat inputként kell megadni, amelyeknek a legenerálására létrehoztam egy egyszerű python scriptet, melynek ha megadjuk a kimeneti kép és bemeneti kép méreteit, akkor az megadja a fix pontos ábrázolását a scale factoroknak.

A számláló egység tehát képes a kimeneti és bemeneti kép pixelein párhuzamosan végiszámolni. Ehhez készítettem egy szimulációt is, amelyen az látható, ahogy egy 512x512-es képet 800x800 ra méretezünk át. Ekkor ha a kimeneti kép közepén vagyunk, akkor a bemeneti kép közepén kell lenni, ami az 3.2 ábrán levő szimuláción jól látszik hogy megtörténik.



3.2. ábra. A bilineáris filter számlálóinak ellenőrzése

Ezután a bemeneti pixel számlálóknak a megfelelő formázása után (egész rész leválasztás) már be lehetett kérni a test bench-től a megfelelő 4 pixel értéket. Miután beérkeztek ezek a pixel értékek, kezdetét vehette a szorzó modul megvalósítása.

3.0.2. Szorzó modul

A szorzó modulban annak megfelelően, hogy hány csatornán érkeznek be a pixelek (1-szürkeárnyalatos, 3-RGB) egy generate blokk segítségével úgy van kiindexelve a bejövő pixel értékekből a megfelelő pixel értékek. Tehát ha 1 csatornáról van szó, akkor a bejövő pixel értékek csak 8 bitesek, de ha 3 csatornás a kép, akkor a bejövő pixelek 24 bitesek. Erre azért van szükség, mert kezdetben csak szürkeárnyalatos képen teszteltem a modult, és később valósítottam meg az RGB képek támogatását is.

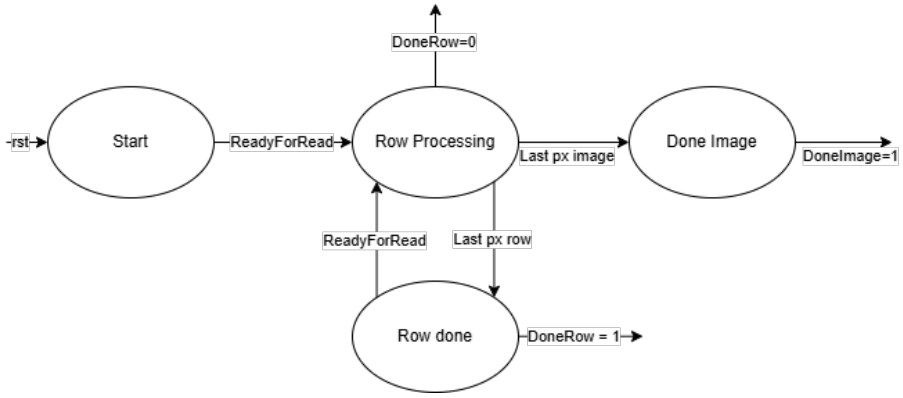
Itt csak megvalósításra kerültek a 2.4 egyenletek szorásai. A kihívás itt viszont azt volt, hogy a szorzások megfelelő méretű regiszterekben legyenek tárolva, ne legyen túlcsordu-

lás, ugyanis ha egy pl Q7.0 pixel értéket megszorzunk egy pl Q1.10 méretű együtthatóval, akkor egy Q8.10 méretű számot kapunk, amit csonkítani kell, és még tovább kell szorozni egy másik Q8.0 méretű számmal (felső és alsó interpoláció eredménye), majd végül a függőleges interpolációt is csonkítani kell. Ezeken kívül továbbá a megfelelő súlyokat kellő mennyiségő órajellel kellett késleltetni, hogy a felső és alsó interpoláció megfelelő súlyokkal dolgozzon, továbbá egy output data valid jelet is elő kellett állapítani, amit szintén megfelelő mennyiségű órajellel kellett késleltetni a beérkező data input valid jelhez képest.

Ezek után már működött a bilineáris szűrő, de annak érdekében hogy a később megvalósítandó sorbufferekkel szinkronban tudjon működni, egy állapotgépbe rendeztem a rendszer működését.

3.0.3. Állapotgép

A modulnak egy egyszerű állapotgépe van, mely a 3.3 ábrán látható.



3.3. ábra. A bilineáris filter állapotgépe

A start állapotben inicializálunk minden változót, countert. Ezután, ha legalább 1 órajelig megkapjuk a readyForRead input jelet, akkor az azt jelenti, hogy a sorbuffer beolvasta a megfelelő sorokat, és készen áll az inputok kiadására. Tehát, ha a kimenetre kiírjuk egy pixelnek a koordinátáit (vagyis az oszlopának a sorszámát elég a sorbuffereknél), akkor az 1 órajel múlva meg fog jelenni a bemeneten.

A processign állapotban egy soron végig halad a filter számlálója, és ezzel párhuzamosan a szorzó egység pár órajel késleltetéssel kiadja a megfelelő kimenő pixel értékét. Ha egy sor végére érünk, akkor a Row done állapotba megyünk át. Ha a kép végére értünk, (sor vége+ oszlopok vége) akkor a done image állapotba térünk.

A Row done állapotban egy soron végigértünk, a doeRow outputot magasra állítjuk, és várjuk hogya readyForRead input magas legyen. Amint ez magasra vált, a doneRow outputot alacsonyra állítjuk, és visszatérünk a Done image állapotba.

Az image Done állapotban a doneimage outputot magasra állítjuk, elkészült a kép. Innen resettel lehet a start állapotba juttatni a modult, és egy újabb kép feldolgozását megkezdeni.

4. fejezet

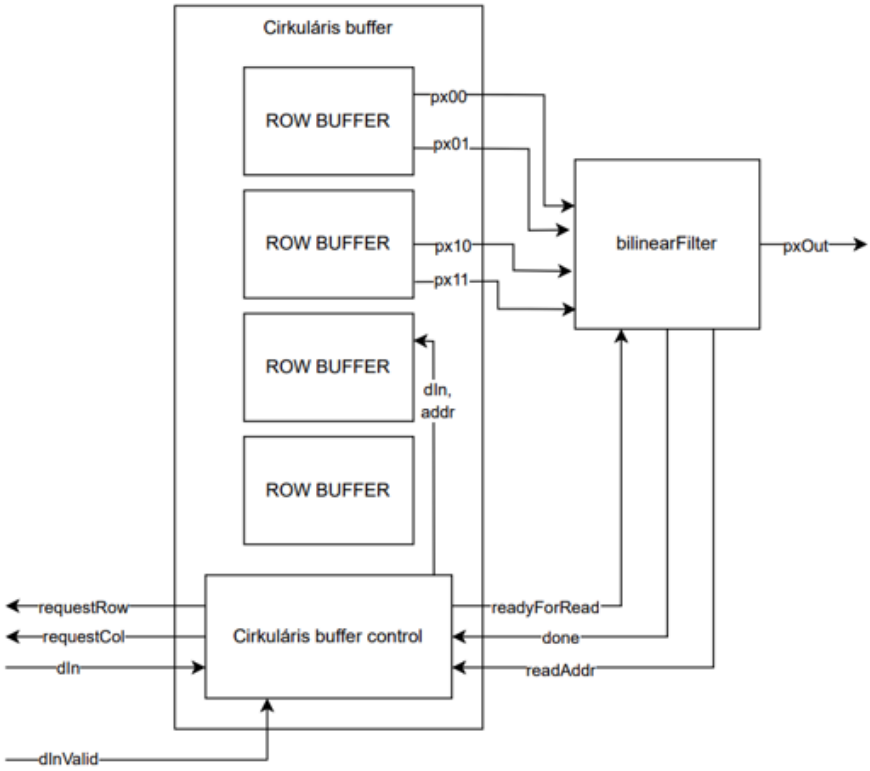
A sorbuffer modul megvalósítása

Eddig úgy működtettem a test benchet a bilineáris filterhez, hogy a bemeneti képet (amit .raw fájlformátumban használtam, annak érdekében hogy pusztán csak a pixel adatokat tartalmazza, és egyéb header információkat ne) beolvastam egy tömbbe, és amikor a filter kért egy pixel értéket, azt csak szimplán kiindexeltem ebből a tömbből a kívánt sor és oszlop alapján. Ez a megoldás tesztelésre jó, de annak érdekében hogy a modul videó folyamot tudjon feldolgozni, ennél robosztusabb megoldásra van szükség. Éppen ezért megalkottam egy sorbuffer modult, aminek az a feladata, hogy eltárolja blokk-ram-okba azokat a sorokat amiket a filter éppen használ (a bilineáris filter esetén egyszerre csak 4 sort használ a szűrő, de polyphase esetén akár 4-et is használhat egyszerre), és ezekből a sorokból pedig már a megfelelő oszlopok pixeleit egyszerűen vissza lehet adni a filter számára.

Továbbá azért fontos még ez a sorbuffer modul, mert majd a továbbiakban, a bejövő video streamet és ezt a sorbuffer modult valamilyen axi interface-el fogom összekapcsolni, amivel a sorbuffer egyszerre tudja írni a blokk ramokat a bejövő video streamből, és közben a filterek számára a megfelelő pixelek értékeit pedig ki tudja adni.

4.0.1. Blokkvázlat

Tekintsük át a sorbuffer magas szintű működését először. Az architektúra blokkvázlata az 4.1 ábrán látható. Mivel a bilineáris szűrő 2 bemeneti sor-t használ egyszerre, ezért ebben az esetben elegendő 4 sornyi blokkramot fenntartani a bufferben. 2 sornyi blokkramból párhuzamosan kiolvassuk a megfelelő oszlopok pixeleinek értékeit, és a maradék 2-be (vagy csak 1 be, vagy 0 ba, erről később) pedig beírjuk a filter által következőleg használt 2 sor értékeit. Tehát egy 2x-es kicsinyítés esetén, a kimeneti kép első sorához a bemeneti kép első két sorát olvassuk a blokkramokból, és közben a bemeneti kép következő 2 sor pixelértékeivel töltjük fel a másik 2 blokkramot, hiszen a kimeneti kép következő sorához, újabb 2 sorra van szükség.



4.1. ábra. A sorbuffer architektúrája

A működés egy cirkuláris bufferre épül, melyben 2^n db blokkram tud elhelyezkedni. Ha a filter k sort használ használ fel a kimeneti pixel meghatározásához, akkor legalább $2k$ blokkramra van szükség a modulban annak érdekében, hogy a legrosszabb esetben is képes legyen a modul egyszerre írni a bejövő adatokat blokkramokba, és a filter tudja olvasni a számára szükséges adatokat.

Tehát mivel a bilineáris filter 2 sort használ, ezért annál $n = 2$, és mivel a polyphase filter pl 4 sort használhat, ezért ott $n = 3$.

A cirkuláris bufferben van egy writePointer, és egy ReadPointer. Amikor egy újabb sort beolvasunk, akkor a writePointert inkrementáljuk, és egy újabb blokkramba olvassuk be a bemeneti adatokat. Amikor újabb sort akarunk koilvasni, akkor pedig a readPointert inkrementáljuk eggyel. Ha túlcsordulna bármelyik pointer, semmi probléma nincsen, a következő blokkram az akkor a 0. lesz akár olvasásról akár írásról van szó. Ezek a pointerek kezelik azt, hogy melyik blokkramból olvasunk ki és melyikbe írunk. Egyszerre csak egy blokkramba írunk, ugyanakkor mivel a bilineáris filternek 2 sorra van szükségea működésre, ezért egyszerre 2 sorból olvasunk. Ez nem probléma, hiszen abból a sorból sosem fogunk olvasni, amelyikbe éppen írunk, vagyis amelyikre éppen a writePointer mutat. Azonban ez az eset a kép legvégén megtörténhet, amikor is már nem olvasunk be több sort, és az utolsó sort olvassuk ki. Ekkor a write pointer és a read pointer ugyan arra a blokkramra mutat. Azonban ilyenkor tudjuk, hogy olvasni akarunk, ezért egy extra változó bevezetésével (forceRead) megoldhatjuk azt a problémát, hogy olvasás valósuljon meg a megadott címre, és ne pedig olvasás.

Ezen kívül az olvasáskor mindig 2 sort kell olvasnunk, azt amelyikre a readpointer mutat, és a következőt. Ezeken felül az oszlopokból mindig azt kell olvasni amelyiket a bilineáris filter kér, (requestCol), és a következőt. Az alábbi kódrészlet azt mutatja be, hogy ez az írás olvasás logika hogyan van megvalósítva.

```
wire [DATA_WIDTH-1:0] ramDataOutA [2**BUFFER_SIZE-1:0];
wire [DATA_WIDTH-1:0] ramDataOutB [2**BUFFER_SIZE-1:0];
//generating the RAM blocks
generate
genvar i;
for(i = 0; i < 2**BUFFER_SIZE; i = i + 1)
begin : ram_generate
ram #(
.DATA_WIDTH(DATA_WIDTH),
.ADDRESS_WIDTH(ORIG_X_SIZE)
) ram_inst_i(
.clk( clk ),

//Port A is written to as well as read from.
//When writing, this port cannot be read from.
.addrA( ((writeSelect == i) && !forceRead && writeEnable) ? requestCol : readAddress ),
.dataA( writeData ),
.weA( ((writeSelect == i) && !forceRead) ? writeEnable : 1'b0 ),
.outA( ramDataOutA[i] ),

//portB is only read from, we are reading the next pixel
.addrB( readAddress + 1'b1 ),
.dataB( writeData ),
.weB( 1'b0 ),
.outB( ramDataOutB[i] )
);
end
endgenerate

//Select which ram to read from
wire [BUFFER_SIZE-1:0]        readSelect0 = readSelect;
wire [BUFFER_SIZE-1:0]        readSelect1 = readSelect+1;

//Steer the output data to the right ports
assign readData00 = ramDataOutA[readSelect0];
assign readData01 = ramDataOutB[readSelect0];
assign readData10 = ramDataOutA[readSelect1];
assign readData11 = ramDataOutB[readSelect1];
```

Mivel a ram egységek amiket példányosítottunk dual port ramok, ezért meg tudjuk csinálni hogy egyszerre 2 pixelt olvasunk ki belőlük. Éppen ezért a második porton csak olvasunk, és az első porton van lekezelve az írás logikája.

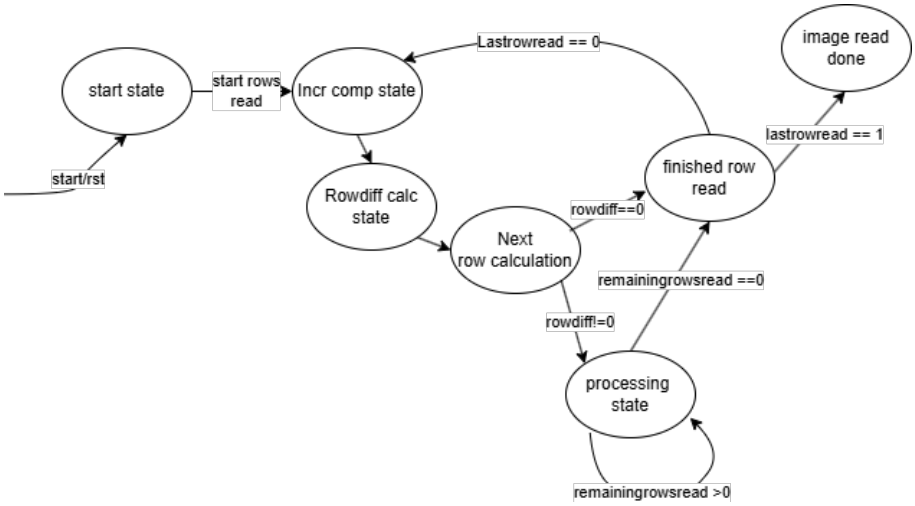
4.0.2. Állapotgép

A sorbuffer modulnak kicsit bonyolultabb állapotgépe van mint a bilinear filter modulnak. Az alapvető gondolat a működése mögött az az, hogy ebben a modulban is jelen van egy számláló, ami a bemeneti képen levő pixel helyzetét számolja (vagyis csak a sorának a helyzetét). Ha ennek a számlálónak az értéke nem billen át egy következő egész számra, (tehát pl 0-ról 0.6666-ra vált az értéke) akkor nem kell újabb sort beolvasnunk ahhoz, hogy a filter számára új adat elérhető legyen, hiszen ugyan azt a két sort fogja használni a következő kimeneti sor meghatározásához, mint amit eddig használt.

Azonban ha a számláló értéke átbillen egy következő egészre (pl 0.666-ról 1.333-ra) akkor a filter következő kimeneti sorának az előállításához már a 0. bemeneti sorra nincs szükség, viszont a 2. bemeneti sorra igen. Ekkor beolvassuk az újabb bemeneti sort, és amikor végez a filter a jelenlegi sor feldolgozásával, a read pointert 1-el fogjuk növelni, ezáltal az outputra az 1. és 2. sor pixelei fognak majd kerülni olvasáskor.

Az utolsó eset, ha a sorszámláló értéke több mint 1 egésszel nőtt. Ekkor lényegtelen hogy mennyi egésszel nőtt az értéke, lehet az 2 is vagy akár 5, mindenképpen 2 új sort kell beolvasnunk, és a read pointert a következő sor feldolgozásánál 2 vel kell növelnünk, hogy a filter számára a 2 frissen beolvasott sor legyen elérhető.

Most, hogy megértettük a működés alapelvét, tekintsük meg az állapotgépet, és nézzük meg mi történik az egyes állapotokban.



4.2. ábra. A sorbuffer állapotgépe

- Start state: itt beolvasásra kerül az első 2 sor, hiszen azt mindenképpen be kell olvasni, ez lesz az első 2 sor amit a filter felhasznál. Amint ez befejeződött, a modul high-ra állítja a readyForRead outputot 1 órajelig, ezzel jelezve a filter modulnak hogy megkezdheti az első sor feldolgozását.
- Incr comp state: Kiszámoljuk a következő sornak az értékét, és az előzőnek az értékét elmentjük egy másik reg-be.
- Rowdiff calc state: Az incr comp state ben kiszámolt és elmentett két regiszter értékének a különbségét vesszük, és meghatározzuk, hogy mennyi egész szám került átlépésre a két sor között. Ezt az értéket a rowDiff regiszterbe mentjük.
- Next row calculation state: Itt az alapján hogy mi volt az előbb meghatározott rowdiff értéke, a következőket csinálhatjuk:
  - rowDiff==0 esetén a következő sorok ugyan azok lesznek amik jelenleg is el vannak mentve, tehát nincs teendő, a finished row read state be ugrunk.
  - rowDiff < NUMROWSTOSTORE mivel a bilineáris filter esetén 2 sort kell elmentenünk, ezért ha a rowDiff az ennél kevesebb (de 0 nál több, tehát 1), akkor a write pointert eggyel növeljük, elmentjük hogy eggyel kevesebb sort kell beolvasnunk (bilineáris filter esetén ez 0 lesz), a remainingRowReads regiszterbe, és azt is elmentjük hogy mennyivel kell majd inkrementálni a read pointert, amikor a filter befejezte az előző sorok feldolgozását. Processing state-be ugrunk.

  - a harmadik esetben 2 vel kell növelni (vagyis annyival amennyi sort eltárolunk a filter működéséhez) a write pointert, és azt is eltároljuk, hogy 2 vel kell megnövelni majd a read pointert. Ezen kívül azt is eltároljuk, hogy mennyi sor olvasása van még hátra (ez a bilineáris filter esetén 1 lesz). Processing state-be ugrunk.

- Processing state: hasonlóan a start state hez, addig olvasunk be sorokat, ameddig be nem olvastuk a megfelelő mennyiségű sorokat, amit az előző state ben meghatároztunk. Ezek után a finished row state be ugrunk.

- finished row state: ha itt vagyunk, az azt jelenti hogy egy sort beolvastunk. Ez tehát azt jelenti, hogy a filter számára elérhetőek a következő sorok amiket használ, attól függetlenül, hogy azok ugyan azok-e vagy különbözőek attól, amiket most használ. Ebben a state ben ha a filter jelez hogy elkészült a sorok feldolgozásával, akkor újra visszaugrunk az incremenet compute state be, és előröl elkezdődik a folyamat, beolvassuk a filter számára szükséges következő sorokat. Ha azonban a kép végére értünk, akkor a finished image state be ugrunk, és a force read regisztert magasba rakjuk, hogy az utolsó sorból is ki tujda olvasni a filter a pixel értékeket.

- finished image state: készen vagyunk, beolvastuk az egész bemeneti képet. Resettel vagy start inputtal újra a start state be hozható az állapotgép

4.0.3. Sorbuffer modul bővítése

Ugyan a sorbuffer modult főként csak a bilineáris filterrel tudtam tesztelni, megpróbáltam egy olyan architektúrát létrehozni, amivel a polyphase filter esetén is használható lesz. Ugyan minden ilyen fejlesztést még nem sikerült implementálnom, de már gondolkodtam rajta hogy miket kellene változtatni.

- Nem lesz elég a filternek az oszlopot megadni hogy melyik pixeleet kéri a bemeneti képből, a sorok is kelleni fognak, hacsak nem bővítem a sorbuffer modul kimeneteit arra, hogy 4 sornak a pixeleit tudja egyszerre kiadni a kimenetre

- polyphase filternél nem azt kell majd itt számon tartani, hogy mi a kimeneti pixel bemeneti képén velő helye, hanem azt, hogy melyik sorban kezdődik a filter ablaka. Erre már előre létrehoztam egy numrowstostore parameter-t, aminek a segítségével ez számítható lesz.

5. fejezet

Eredmények, a projekt helyzete

A projekt jelenlegi állása szerinte képes a bilineáris filterrel és a sorbufferekkel kiegészített modul segítségével tetszőlegesen átméretezni képeket. Mindkét modulhoz (sorbuffer modul, és bilineáris filter modul) készült testbench, amikkel azok működése ellenőrizhetővé vált. Ezen kívül elkészült egy harmadik testbench is, amiben a két modul integrálva van, és egymáshoz van csatlakoztatva. Néhány próba kép a F.1 fejezetben található, ahol az eredeti 256x256 lena képet nagyítottam fel, majd egy tetszőleges méretűre méreteztem át.

A polyphase filter idő szűkében sajnos nem került még megvalósításra, még jelenleg fejlesztés alatt van, de mivel az irodalomkutatás része már megvan, ezért azta jövőben jóval egyszerűbb lesz megvalósítani, mint akár a félév elején az egyszerűbb bilineáris filtert. Ezt lehet az első továbbfejlesztésnek nevezni.

Ezen kívül a projekt jelenleg csak standalone képeket tud átméretezni, video folyamot nem, úgyhogy ezt a fejlesztést, hogy valamilyen axi interface-t csatlakoztatunk a sorbuffer modulhoz, és egy bemeneti video streamet juttatunk el annak, lehetne a második nagyobb fejlesztésnek is nevezni.

Harmadikként pedig a sorbuffer modul kisebb átdolgozsára szorul, hogy a polyphase filterrel is működni tudjon, de ezt már annak a fejezetében is említettem. A fejlesztés során próbáltam gondolni arra, hogy kicsit általánosabban működjön a sorbuffer modul, de mivel nem volt polyphase filter modulom elkészülve, amivel tesztelni tudtam volna, ezért ezt nem is tudtam 100% ban így fejleszteni.

A projekt során rengeteget tanultam, mind a verilog nyelvből, mind FPGA működéséről, (hiszen ezekkel most találkoztam először) mind pedig általános képfeldolgozásról. Több külső eszközt kellett használnom hogy ellenőrizzem az eredményeim, ilyenek pl az IrfanView a .raw képek megtekintésére, vagy az ImHex hex editor, ahol a képeket pixel érték szinten tudtam vizsgálni. Ezen kívül segéd python scripteket is kellett írnom a modul teszteléséhez pl a scale factor megállapításához. A projekt forráskódja megtalálható a saját Githubomon, az alábbi linken: `https://github.com/NuunMoon/FPGA_scaler`

Irodalomjegyzék

[1] URL https://www.intel.com/content/www/us/en/docs/programmable/683329/24-1/scaler.html.

[2] URL https://www.xilinx.com/products/intellectual-property/1-1bkvlcw.html.

[3] URL https://www.zipcores.com/digital-video-scaler.html.

[4] URL https://docs.amd.com/v/u/en-US/pg009_v_scaler.

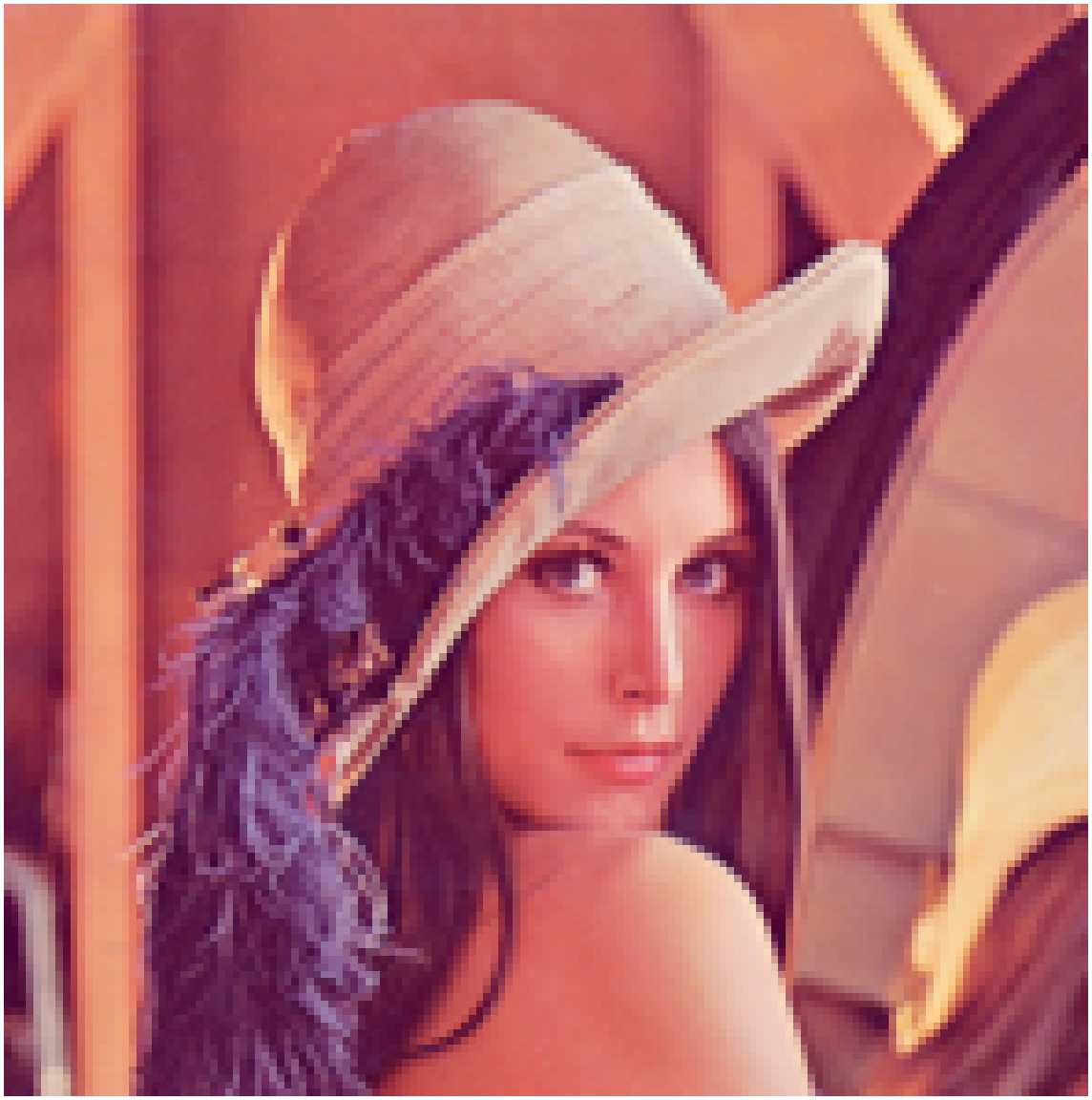
[5] 2024. Feb. URL https://en.wikipedia.org/wiki/Bilinear_interpolation.

[6] Long Chen – Wen Tang – Nigel John: Self-supervised monocular image depth learning and confidence estimation. *Neurocomputing*, 381. évf. (2018. 03).

[7] Alwyn Mathew: Forward mapping, 2019. Feb.
URL https://alwynm.github.io/blog/research/forward-mapping.

Függelék

F.1. Próba képek



F.1.1. ábra. Lena eredeti változata, 256x256 méretű

F.1.2. ábra. Lena 2x felnagyított változata, 512x512 méretű

F.1.3. ábra. Lena tetszőlegesen nagyított változata, 800x150 méretű